UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

Bancos de Dados

Notas de Aula

Francisco Rapchan
rapchan@writeme.com

Capítulo 1 INTRODUÇÃO

Neste capítulo faremos uma pequena introdução aos conceitos de bancos de dados.

1.1 O QUE SÃO BANCOS DE DADOS?

Há muitas definições para bancos de dados. Cada autor costuma dar uma definição:

- Segundo [**Date**], o sistema de banco de dados consiste em um sistema de manutenção de informações por computador que tem por objetivo manter as informações e disponibilizá-las aos seus usuários quando solicitadas.
- Segundo [**Palmer**], um banco de dados é uma coleção de dados, organizados e integrados, que constituem uma representação natural de dados, sem imposição de restrições ou modificações para todas as aplicações relevantes sem duplicação de dados.

Há autores que fazem distinção entre **Dado e Informação**.

- Dado: refere-se aos valores fisicamente registrados no banco de dados.
- Informação: refere-se ao significado destes valores para determinado usuário.

1.2 SGBD - SISTEMA DE GERENCIAMENTO DE BANCOS DE DADOS

Um SGBD - Sistema de Gerenciamento de Banco de Dados, é uma coleção de programas que permitem ao usuário definir, construir e manipular Bases de Dados para as mais diversas finalidades.

De forma resumida, podemos dizer que:

SGBD = Coleção de dados inter-relacionados + Conjunto de programas para acessá-los

Os principais objetivos de um SGBD é:

- Prover um ambiente que seja adequado e eficiente para recuperar e armazenar informações de bancos de dados.
- Prover os usuários com uma visão abstrata dos dados.

1.3 SGBD x GA

Não devemos confundir um Sistema Gerenciador de Banco de Dados com um Gerenciador de Arquivos.

Desvantagens de um GA frente a um SGBD:

- **Redundância e inconsistência de dados.** Arquivos com formatos diferentes, diferentes linguagens de programação, elementos de informação duplicados em diversos arquivos;
- **Dificuldade no acesso aos dados.** Dados recuperados de forma inconveniente e ineficiente;
- **Isolamento de dados.**
- **Anomalias de acesso concorrente.** dados acessados por diferentes programas aplicativos → supervisão difícil ;
- **Problemas de segurança.** Difícil definição de visibilidade para usuários;
- **Problemas de integridade.** Restrição de integridade nos valores dos atributos.

Vamos definir algumas regras básicas e claras para um sistema de manipulação de dados ser considerado um SGBD. Fica implícito que se ao menos uma das características abaixo não estiver presente no nosso "candidato" a SGBD, este poderá ser um GA (Gerenciador de Arquivo) de altíssima qualidade, "quase" um SGBD, mas não um SGBD [Nicochelli].

- **Regra 1: Auto-Contenção-** Um SGBD não contém apenas os dados em si, mas armazena completamente toda a descrição dos dados, seus relacionamentos e formas de acesso. Normalmente esta regra é chamada de *Meta-Base* de Dados. Em um GA, em algum momento ao menos, os programas aplicativos declaram estruturas (algo que ocorre tipicamente em C, COBOL e BASIC), ou geram os relacionamentos entre os arquivos (típicos do ambiente xBase). Por exemplo, quando você é obrigado a definir a forma do registro em seu programa, você não está lidando com um SGBD.
- **Regra 2: Independência dos Dados**- Quando as aplicações estiverem realmente imunes a mudanças na estrutura de armazenamento ou na estratégia de acesso aos dados, podemos dizer que esta regra foi atingida. Portanto, nenhuma definição dos dados deverá estar contida nos programas da aplicação. Quando você resolve criar uma nova forma de acesso, um novo índice, *se precisar alterar o código de seu aplicativo*, você não está lidando com um SGBD.
- **Regra 3: Abstração dos Dados**- Em um SGBD real é fornecida ao usuário somente uma representação conceitual dos dados, o que não inclui maiores detalhes sobre sua forma de armazenamento real. O chamado Modelo de Dados é um tipo de abstração utilizada para fornecer esta representação conceitual. Neste modelo, um esquema das tabelas, seus relacionamentos e suas chaves de acesso são exibidas ao usuário, porém nada é afirmado sobre a criação dos índices, ou como serão mantidos, ou qual a relação existente entre as tabelas que deverá ser mantida íntegra. Assim se você desejar inserir um pedido em um cliente inexistente e esta entrada não for automaticamente rejeitada, você não está lidando com um SGBD.
- **Regra 4: Visões**- Um SGBD deve permitir que cada usuário visualize os dados de forma diferente daquela existente previamente no Banco de Dados. Uma visão consiste de um subconjunto de dados do Banco de Dados, necessariamente derivados dos existentes no Banco de Dados, porém estes não deverão estar explicitamente armazenados. Portanto, toda vez que você é obrigado a replicar uma estrutura, para fins de acesso de forma diferenciada por outros aplicativos, você não está lidando com um SGBD.
- **Regra 5: Transações**- Um SGBD deve gerenciar completamente a integridade referencial definida em seu esquema, sem precisar em tempo algum, do auxílio do programa aplicativo. Desta forma exige-se que o banco de dados tenha ao menos uma instrução que permita a gravação de uma série modificações simultâneas e uma instrução capaz de cancelar um série modificações. Por exemplo, imaginemos que estejamos cadastrando um pedido para um cliente, que este deseje reservar 5

itens de nosso estoque, que estão disponíveis e portanto são reservados, porém existe um bloqueio financeiro (duplicatas em atraso) que impede a venda. A transação deverá ser desfeita com apenas uma instrução ao Banco de Dados, sem qualquer modificações suplementares nos dados. Caso você se obrigue a corrigir as reservas, através de acessos complementares, você não está lidando com um SGBD.

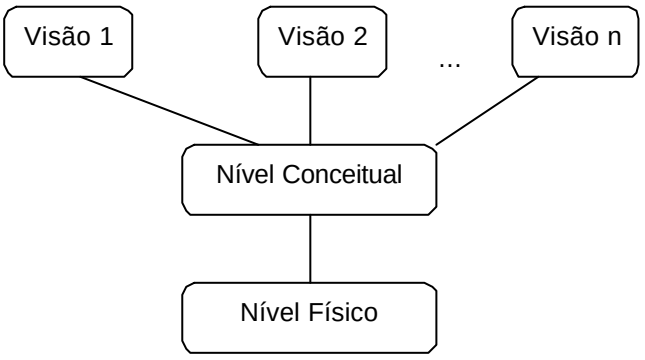
- **Regra 6: Acesso Automático**- Em um GA uma situação típica é o chamado *deadlock*, o abraço mortal. Esta situação indesejável pode ocorrer toda vez que um usuário travou um registro em uma tabela e seu próximo passo será travar um registro em uma tabela relacionada à primeira, porém se este registro estiver previamente travado por outro usuário, o primeiro usuário ficará paralisado, pois, estará esperando o segundo usuário liberar o registro em uso, para que então possa travá-lo e prosseguir sua tarefa. Se por hipótese o segundo usuário necessitar travar o registro travado pelo primeiro usuário (!), afirmamos que ocorreu um abraço mortal, pois cada usuário travou um registro e precisa travar um outro, justamente o registro anteriormente travado pelo outro! Imaginemos um caso onde o responsável pelos pedidos acabou de travar o Registro Item de Pedido, e, necessita travar um registro no Cadastro de Produtos, para indicar uma nova reserva. Se ao mesmo tempo estiver sendo realizada uma tarefa de atualização de pendências na Tabela de Itens, e para tanto, previamente este segundo usuário travou a Tabela de Produtos, temos a ocorrência do abraço mortal. Se a responsabilidade de evitar esta ocorrência for responsabilidade da aplicação, você não está lidando com um SGBD.

1.4 ABSTRAÇÃO DE DADOS

Os SGBD's fornecem aos usuários uma visão abstrada dos dados. Isto significa que os usuários não precisam saber como os dados são armazenados e mantidos.

Níveis de abstração de dados:

- **Nível Físico (ou interno)**: como os dados estão realmente armazenados (complexas estruturas de dados).
- **Nível Conceitual:** visão do conjunto de usuários.
- **Nível de Visões (ou externo):** mais alto nível de abstração. Descreve apenas parte do banco de dados.



1.5 MODELOS DE DADOS - NÍVEIS: CONCEITUAL E VISUAL

Ferramentas conceituais para descrição de dados, relacionamentos de dados, restrições de integridade;

Grupos:

- Modelos lógicos baseados em objetos → Modelo ER
  Modelo Orientado a Objetos;
- Modelo Lógico baseado em Registros;

Derivação:

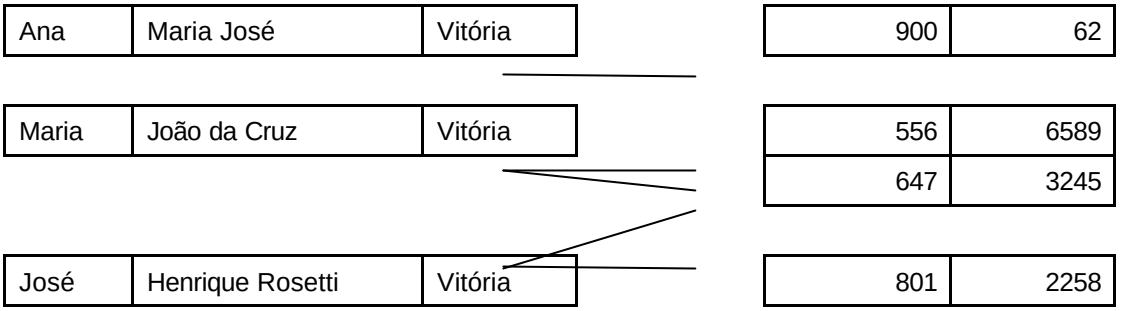
Modelos de Banco de Dados → Relacional, Rede e hierárquico + Recente (Modelo OO);

1.6 MODELO RELACIONAL

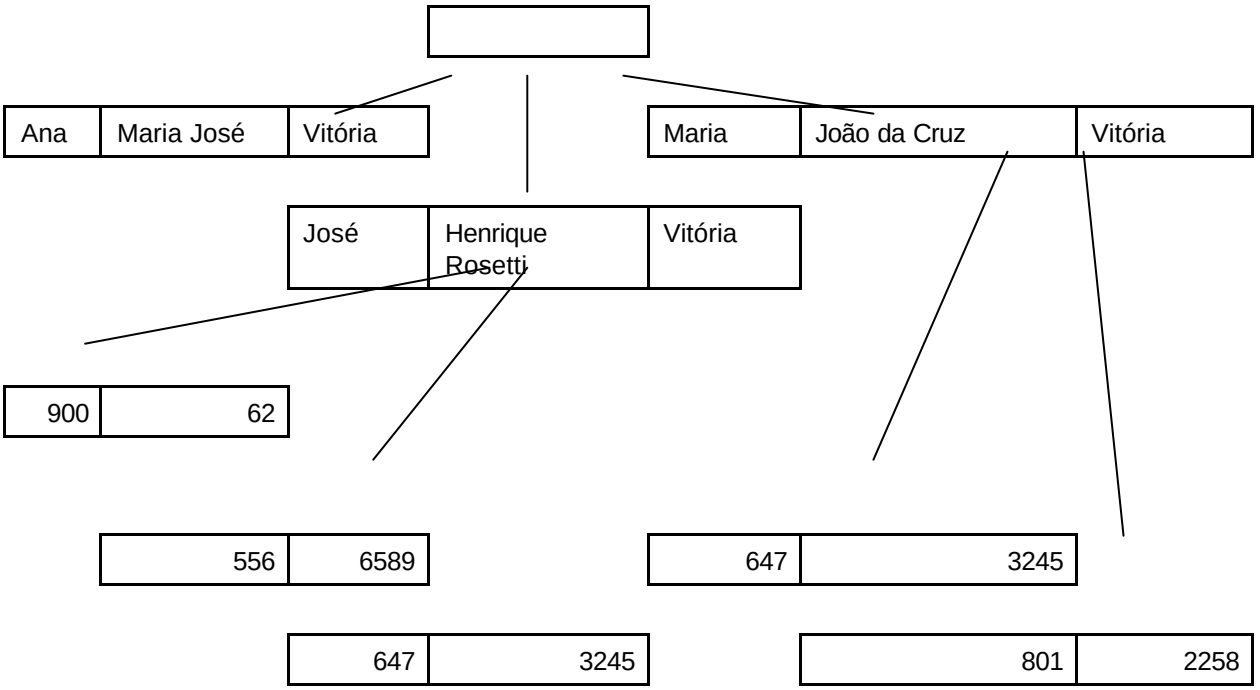
nome	rua	cidade	número
Ana	Maria José	Vitória	900
Maria	João da Cruz	Vitória	556
Maria	João da Cruz	Vitória	647
José	Henrique Rosetti	Vitória	801
José	Henrique Rosetti	Vitória	647

número	saldo
900	62
556	6589
647	3245
801	2258

1.7 MODELO REDE



1.8 MODELO HIERÁRQUICO



1.9 INSTÂNCIAS E ESQUEMAS

Instâncias → Coleção de registros de um banco de dados em um dado momento;

Esquemas → Projeto geral do banco de dados → Físico, Conceitual, Subesquemas;

1.10 INDEPENDÊNCIA DE DADOS

Habilidade de modificar um esquema em um nível sem afetar um nível + alto;

- Independência física de dados → Muda o esquema físico, não reescreve aplicações;
- Independência lógica de dados → Muda o esquema conceitual, não reescreve aplicações;

Qual abordagem é mais fácil de ser alcançada?

1.11 LINGUAGEM DE DEFINIÇÃO DE DADOS (DDL)

Contém a especificação dos esquemas de banco;

1.12 LINGUAGEM DE MANIPULAÇÃO DE DADOS (DML)

São responsáveis pela:

- Recuperação da informação armazenada no banco de dados;
- Inserção de novos dados nos bancos de dados;
- Eliminação de dados nos bancos de dados;
- Modificação de dados armazenados no banco de dados;

1.13 TAREFAS DE UM SGBD

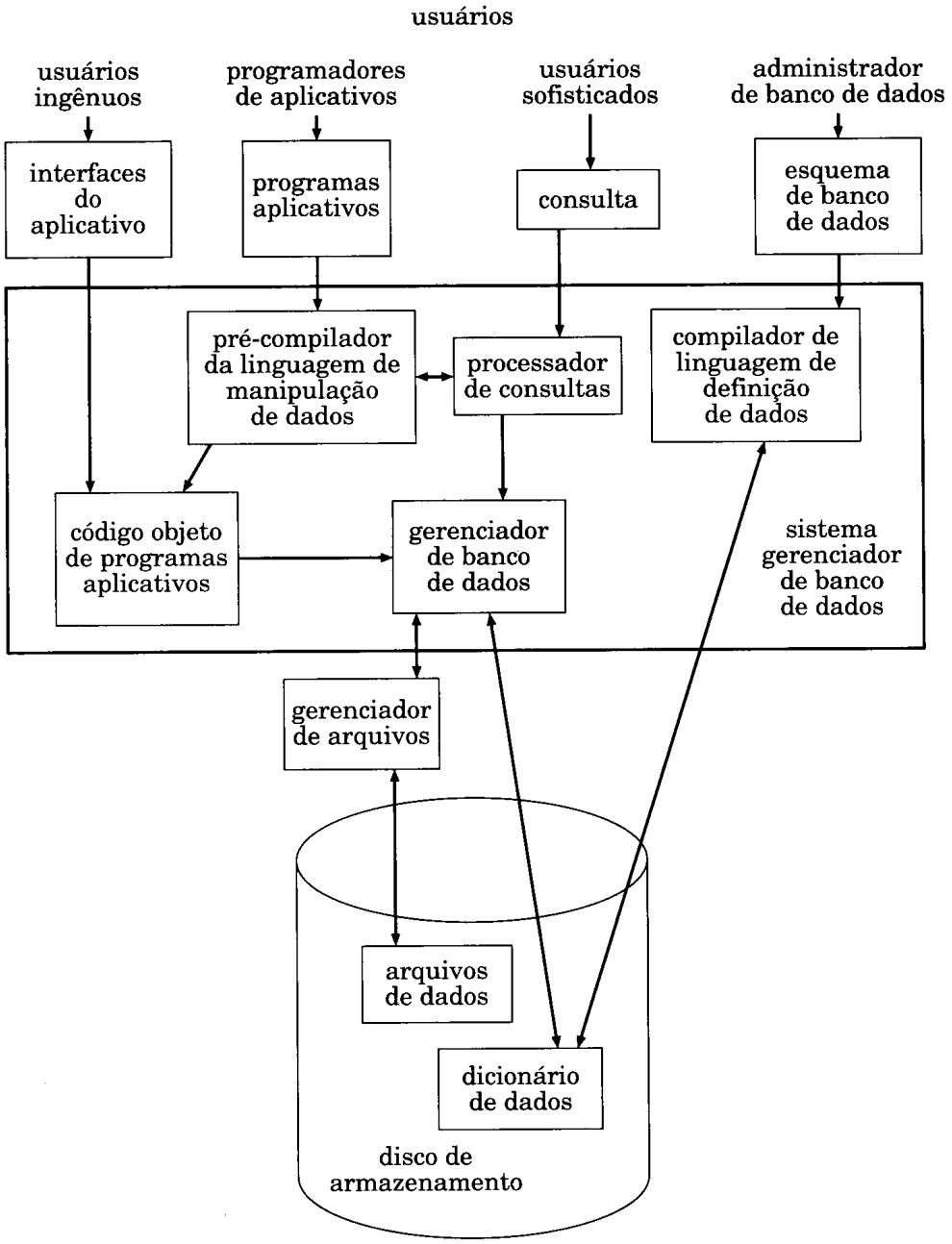
- Interação com o gerenciador de arquivos;
- Cumprimento da integridade;
- Cumprimento da segurança;
- Cópias de segurança e recuperação;
- Controle de concorrência.

1.14 TAREFAS DE UM ADMINISTRADOR DE BANCO DE DADOS (DBA)

- Definição de esquema;
- Definição de estrutura de armazenamento e método de acesso;
- Modificação de esquema e organização física;
- Concessão de autorização para acesso aos dados;
- Especificação de restrição de integridade;

1.15 ESTRUTURA GERAL DO SISTEMA

- Gerenciador de arquivos;
- Gerenciador de banco de dados;
- Processador de consultas;
- Compilador DML;
- Compilador DDL;



Capítulo 2 MODELO E ÁLGEBRA RELACIONAL

O modelo relacional foi apresentado por E. F. Codd em um clássico artigo intitulado *"A Relacional Model for Large Shared Data Banks"* na *Comunications orf the ACM* em Junho de 1970. Codd também foi o responsável por publicar os primeiros trabalhos sobre álgebra relacional nos primeiros anos da década de 70.

Este capítulo mostra os fundamentos do Modelo Relacional e faz uma introdução dos principais aspectos da Álgebra Relacional.

2.1 CONCEITOS DO MODELO RELACIONAL

Segundo [Date], um banco de dados relacional é um banco de dados que o usuário percebe como uma coleção de relações (TABELAS) normalizadas, de vários graus, que se modificam ao longo do tempo.

O Modelo de Dados relacional representa os dados contidos em um Banco de Dados através de relações. Estas relações contém informações sobre as entidades representadas e seus relacionamentos. O Modelo Relacional, é claramente baseado no conceito de matrizes, onde as chamadas linhas (das matrizes) seriam os registros e as colunas (das matrizes) seriam os campos. Os nomes das tabelas e dos campos são de fundamental importância para nossa compreensão entre o que estamos armazenando, onde estamos armazenando e qual a relação existente entre os dados armazenados.

2.1.1 Relações

Cada linha de nossa relação será chamada de *TUPLA* e cada coluna de nossa relação será chamada de *ATRIBUTO*. O conjunto de valores passíveis de serem assumidos por um atributo, será intitulado de *DOMÍNIO.*

Um banco de dados relacional consiste em uma coleção de tabelas.

- Cada tabela é formada por linhas e colunas. Dizemos que uma tabela é uma relação de tuplas.
- Cada linha representa um relacionamento entre um conjunto de valores. Dizemos que cada linha é uma tupla.
- Cada coluna da tabela representa um atributo da mesma.
- O GRAU DA RELAÇÃO é o número de atributos na relação, ou o que seria equivalente, o número de domínios básicos.

2.1.2 Domínio de um atributo

O domínio de um atributo é um conjunto de valores atômicos que se aplicam ao atributo. Dito de outra forma, domínio é um grupo de valores, a partir dos quais um ou mais atributos (colunas) retiram seus valores reais.

Um atributo em uma tupla só pode representar um valor, não pode representar um conjunto de valores. Por exemplo um atributo telefone que represente o telefone de um cliente não pode receber mais de um valor (mesmo que o cliente tenha mais de um telefone).

Dizemos então que o atributo não pode ser multivalorado.

2.1.3 Esquema relacional de banco de dados

O esquema de uma relação, nada mais são que os campos (colunas) existentes em uma tabela. Já a instância da relação consiste no conjunto de valores que cada atributo assume em um determinado

instante. Portanto, os dados armazenados no Banco de Dados, são formados pelas instâncias das relações.

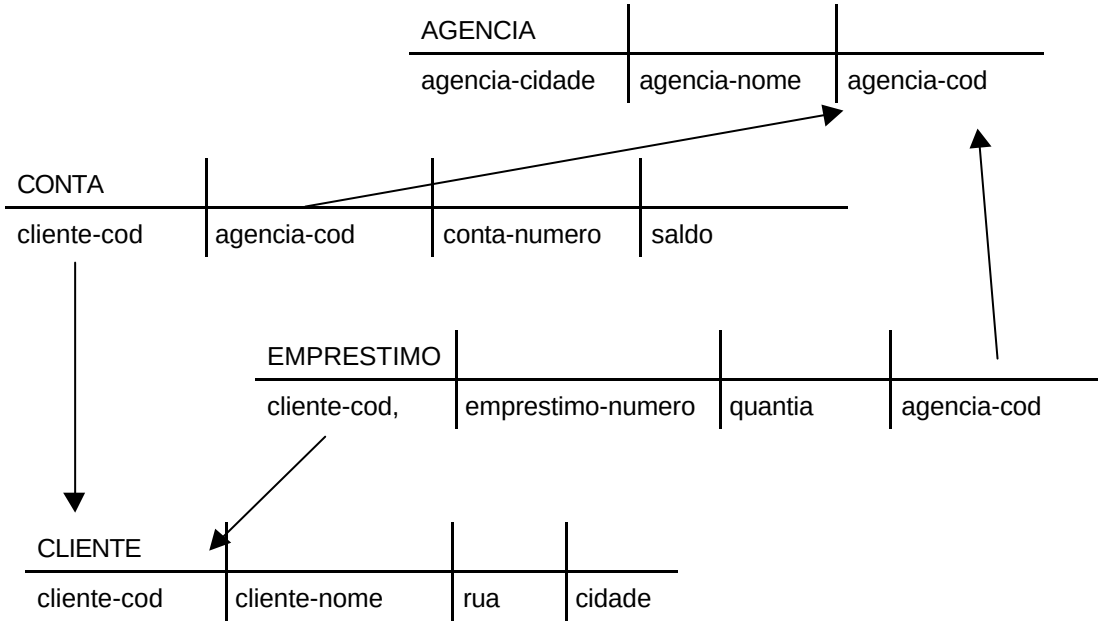
O esquema relacional pode ser representado de várias formas.

Exemplo

Esquema Banco:

AGENCIA = (agencia-cod, agencia-nome, agencia-cidade)
CLIENTE = (cliente-cod, cliente-nome, rua, cidade)
CONTA = (agencia-cod, conta-numero, cliente-cod, saldo)
EMPRESTIMO = (agencia-cod, cliente-cod, emprestimo-numero, quantia)

De uma forma visual:



Observe que na representação gráfica há uma série de linhas direcionadas. Elas indicam a origem das chaves estrangeiras em cada tabela. Assim. *CONTA*.c*liente-code* é chave estrangeira vinda de *CLIENTE.cliente-cod*.

2.1.4 Propriedades das relações

- As tuplas de uma relação não têm uma ordem definida.
- Uma tupla é formada por um conjunto de pares (<atributo>, <valor>) e também não têm uma ordem explícita.
- Todos os valores são atômicos, não há atributos multivalorados (chamamos esta regra de primeira forma normal).
- Não há tuplas duplicadas. Não pode haver duas linhas com os mesmos valores em uma tabela.
- Cada chave candidata deve ser única para cada tupla.
- Nenhuma chave primária pode ser nula.
- Uma tupla em uma relação que referencia outra relação deve referenciar uma tupla existente naquela relação (chamamos esta regra de integridade referencial).

2.2 ÁLGEBRA RELACIONAL

A álgebra relacional é uma linguagem de consulta procedural. Ela consiste em um conjunto de operações que tornam uma ou duas relações como entrada e produzem uma nova relação como resultado.

As operações fundamentais na álgebra relaciona são: selecionar, projetar, renomear, (unárias) - produto cartesiano, união e diferença de conjuntos (binárias).

Além das operações fundamentais, existem outras operações: interseção de conjuntos, ligação natural, dentre outras, que são definidas em termos das operações fundamentais.

2.2.1 Operação selecionar

Seleciona tuplas que satisfazem um dado predicado (condição), descartando as outras. Usamos a letra minúscula grega sigma σ para representar a seleção. O predicado aparece subscrito em σ. A relação argumento aparece entre parênteses seguindo o σ.

A forma geral de uma seleção é:

$$\sigma_{\text{<condições>}} (\text{RELAÇÃO})$$

As comparações são permitidas usando $=, \neq, <, \leq, >$ e $\geq$ e os conectivos e (^) e ou ($\vee$) e que envolvam apenas os atributos existentes em na RELAÇÃO.

Exemplo

Selecione as tuplas da relação empréstimo onde o código da agência é 0662.

$$\sigma_{\text{agencia-cod=0662}}(\text{EMPRESTIMO})$$

Exemplo

Encontrar todas as tuplas onde a quantia emprestada seja maior que 1200.

$$\sigma_{\text{quantia >1200}}(\text{EMPRESTIMO})$$

2.2.2 Operação projetar

A operação projetar é uma operação unária que retorna sua relação argumento, com certas colunas deixadas de fora. A projeção é representada pela letra grega (π).

A forma geral é:

$$\pi_{\text{<atributos da relação>}} (\text{RELAÇÃO})$$

Exemplo

Mostre o código dos clientes e das agências nas quais eles tomaram empréstimos.

$$\pi_{\text{agencia-cod,cliente-cod}}(\text{EMPRESTIMO})$$

Exemplo

Mostre os clientes que moram em “Aracruz”.

Devemos fazer uma seleção de todos os clientes que moram em Aracruz, em seguida projetar o código destes clientes.

$$\pi_{\text{ cliente-nome}} (\sigma_{\text{cliente-cidade=”Aracruz”}} (\text{CLIENTE}))$$

Observe que neste caso, se houver clientes com o mesmo nome, apenas um nome aparecerá na tabela resposta!

2.2.3 Operação produto cartesiano

Esta operação combina atributos (colunas) a partir de diversas relações. Trata-se de uma operação binária muito importante. Esta operação nos mostra todos os atributos das relações envolvidas.

A forma geral é:

RELAÇÃO1 X RELAÇÃO2

Exemplo

Para selecionarmos todos os nomes dos clientes que possuam empréstimo na agência cujo código é 0662, escrevemos:

$\pi_{\text{cliente-nome}} (\sigma_{\text{agencia-cod=0662 ^ Emprestimo.cliente-cod = CLIENTE. cliente-cod}} (\text{EMPRESTIMO X CLIENTE}))$

Deve-se tomar cuidado com ambigüidades no nomes dos atributos.

Exemplo

Esquema CANTINA-DO-VILMO:

CLIENTE = (codigo, nome)

FIADO = (codigo, valor)

Se quisermos saber os nomes dos clientes e seus fiados, deveríamos fazer:

$\pi_{\text{codigo,nome,valor}} (\sigma_{\text{CLIENTE.codigo=FIADO.codigo}} (\text{Cliente x Fiado}))$

Supondo as tabelas CLIENTE e FIADO abaixo, podemos fazer uma representação das etapas desta consulta.

CLIENTE

codigo	nome
1	Chico
2	Ana

FIADO

codigo	valor
1	5,00
1	10,00
2	4,00
2	8,00

CLIENTE X FIADO

CLIENTE.codigo	nome	FIADO.codigo	valor
1	Chico	1	5,00
1	Chico	1	10,00
1	Chico	2	4,00
1	Chico	2	8,00
2	Ana	1	5,00
2	Ana	1	10,00
2	Ana	2	4,00
2	Ana	2	8,00

($\sigma_{CLIENTE.codigo=FIADO.codigo}$ (Cliente x Fiado)

CLIENTE.codigo	nome	FIADO.codigo	valor
1	Chico	1	5,00
1	Chico	1	10,00
2	Ana	2	4,00
2	Ana	2	8,00

2.2.4 Operação Renomear

A operação de renomear uma tabela é usada sempre que uma relação aparece mais de uma vez em uma consulta. É representada pela letra grega ρ .

A forma geral é:

$$\rho_{<novo\ nome>}(RELAÇÃO2)$$

Uma outra forma de renomear uma RELAÇÃO é atribui-la a uma outra. Isto é feito com o símbolo ←.

RELAÇÃO2 ← RELAÇÃO1

Exemplo

Encontre todos os clientes que moram na mesma rua e cidade que João.

Podemos obter a rua e a cidade de João da seguinte forma:

$$t \leftarrow \pi_{rua,\ cidade}(\sigma_{cliente\_nome="João"}(CLIENTE))$$

Entretanto, para encontrar outros clientes com esta rua e cidade, devemos referir-nos à relação Clientes pela segunda vez. Perceba que se for inserida novamente uma relação clientes na consulta gerará ambiguidade. Por isso, devemos renomeá-la.

$$\rho_{CLIENTE2}(CLIENTE)$$

Obtemos então:

$$\pi_{rua,\ cidade}(\sigma_{CLIENTE.cidade = CLIENTE2.cidade\ \wedge\ CLIENTE.rua = CLIENTE2.rua}(t\ X\ \rho_{CLIENTE2}(CLIENTE))$$

CLIENTE

clinte-cod	cliente-nome	rua	cidade
1	Maria	Rua 1	Cariacica
2	João	Rua 2	Vitória
3	Ana	Rua 3	Cariacica
4	José	Rua 2	Vitória

CLIENTE X CLIENTE2

clinte-cod	cliente-nome	rua	cidade	cliente2-cod	cliente2-nome	rua	Cidade
1	Maria	Rua 1	Cariacica	1	Maria	Rua 1	Cariacica
1	Maria	Rua 1	Cariacica	2	João	Rua 2	Vitória
1	Maria	Rua 1	Cariacica	3	Ana	Rua 3	Cariacica
1	Maria	Rua 1	Cariacica	4	José	Rua 2	Vitória
2	João	Rua 2	Vitória	1	Maria	Rua 1	Cariacica
2	João	Rua 2	Vitória	2	João	Rua 2	Vitória
2	João	Rua 2	Vitória	3	Ana	Rua 3	Cariacica
2	João	Rua 2	Vitória	4	José	Rua 2	Vitória
3	Ana	Rua 3	Cariacica	1	Maria	Rua 1	Cariacica
3	Ana	Rua 3	Cariacica	2	João	Rua 2	Vitória
3	Ana	Rua 3	Cariacica	3	Ana	Rua 3	Cariacica
3	Ana	Rua 3	Cariacica	4	José	Rua 2	Vitória
4	José	Rua 2	Vitória	1	Maria	Rua 1	Cariacica
4	José	Rua 2	Vitória	2	João	Rua 2	Vitória
4	José	Rua 2	Vitória	3	Ana	Rua 3	Cariacica
4	José	Rua 2	Vitória	4	José	Rua 2	Vitória

2.2.5 Operação União (binária)

A operação binária união é representada, como na teoria dos conjuntos, pelo símbolo **È**.

A forma geral é:

RELAÇÃO1 **È** RELAÇÃO2

Suponha que quiséssemos saber todas as pessoas que possuem CONTA ou EMPRÉSTIMO numa determinada agência. Com os recursos que temos até agora, não seria possível conseguirmos tal informação. Nessa situação, deveríamos fazer a união de todos que possuem conta com todos que possuem empréstimos nessa agência.

Exemplo

Selecionar todos os clientes que possuam conta ou empréstimos ou ambos na agência 051.

$$\pi_{\text{cliente-nome}} (\sigma_{\text{agencia-cod=”051” ^ CONTA.cliente-cod = CLIENTE.cliente-cod}}(\text{CONTA X CLIENTE}))$$

$$\cup$$

$\pi_{\text{cliente-nome}}$ ($\sigma_{\text{agencia-cod=”051” ^ CONTA.cliente-cod = CLIENTE.cliente-cod}}$(EMPRESTIMO X CLIENTE))

Uma vez que as relações são conjuntos, as linhas duplicadas são eliminadas.

Para uma operação União r ∪ s ser válida, necessitamos que duas condições devem ser cumpridas:

- As relações r e s precisam ter a mesma paridade. Isto é, elas precisam ter o mesmo número de atributos;
- Os domínios do i-ésimo atributo de r e do i-ésimo atributo de s devem ser os mesmos.

2.2.6 A operação diferença de conjuntos

A operação diferença de conjuntos permite-nos encontrar tuplas que estão em uma relação e não em outra. A expressão r - s resulta em uma relação que contém todas as tuplas que estão em r e não em s. A forma geral é:

RELAÇÃO1 **-** RELAÇÃO2

Exemplo

Encontrar todos os clientes que possuam uma conta mas não possuem um empréstimo na agência 051.

$\pi_{\text{cliente-nome}}$ ($\sigma_{\text{agencia-cod=”051” ^ CONTA.cliente-cod = CLIENTE.cliente-cod}}$ (CONTA X CLIENTE))

-

$\pi_{\text{cliente-nome}}$ ($\sigma_{\text{agencia-cod=”051” ^ CONTA.cliente-cod = CLIENTE.cliente-cod}}$ (EMPRESTIMO X CLIENTE))

Observação: A operação diferença não é comutativa. Em geral r – s ≠ s – r .

Exemplo

Suponha o seguinte esquema:

```
Esquema CANTINA DO VILMO:
      CLIENTE = (codigo, nome)
      FIADO = (codigo, valor)
```

Mostre a conta de fiado mais alta.

MENORES ← π_{valor} ($\sigma_{\text{valor < AUX.valor}}$ (FIADO X ρ_{AUX} (FIADO))

MAIOR ← π_{valor} (FIADO) - MENORES

CLIENTE

codigo	nome
1	Chico
2	Ana

FIADO

codigo	valor
1	5,00
1	10,00
2	4,00
2	8,00

FIADO X r_{AUX}(FIADO)

codigo	valor	AUX.codigo	AUX.valor
1	5,00	1	5,00
1	*5,00*	*1*	*10,00*
1	5,00	2	4,00
1	*5,00*	*2*	*8,00*
1	10,00	1	5,00
1	10,00	1	10,00
1	10,00	2	4,00
1	10,00	2	8,00
2	*4,00*	*1*	*5,00*
2	*4,00*	*1*	*10,00*
2	4,00	2	4,00
2	*4,00*	*2*	*8,00*
2	8,00	1	5,00
2	*8,00*	*1*	*10,00*

2.2.7 Operação interseção de conjuntos

É representado pelo símbolo $\cap$.

A forma geral é:

$$\text{RELAÇÃO1} \cap \text{RELAÇÃO2}$$

Exemplo

Encontrar todos os clientes com um empréstimo e uma conta na agência “051”.

$$\pi_{\text{cliente-nome}} (\sigma_{\text{agencia-cod=”051” ^ CONTA.cliente-cod = CLIENTE.cliente-cod}}(\text{CONTA X CLIENTE}))$$

$$\cap$$

$$\pi_{\text{cliente-nome}} (\sigma_{\text{agencia-cod=”051” ^ CONTA.cliente-cod = CLIENTE.cliente-cod}}(\text{EMPRESTIMO X CLIENTE}))$$

A operação Interseção de conjuntos pode ser expressa em função das operações fundamentais da seguinte forma: $r \cap s = r - (r – s)$.

2.2.8 Operação Ligação natural

A ligação natural é uma operação binária que permite combinar certas seleções e um produto cartesiano em uma única operação. É representada pelo símbolo |X|.

A operação ligação natural forma um produto cartesiano de seus dois argumentos, faz uma seleção forçando uma equidade sobre os atributos que aparecem em ambos os esquemas relação.

A forma geral é:

$$\text{RELAÇÃO1} \; |x|_{(\text{atributoA, atributoB})} \; \text{RELAÇÃO2}$$

Que equivale a:

$$\sigma_{\text{atributoA = atributoB}} (\text{RELAÇÃO1 X RELAÇÃO2})$$

Exemplo

Encontre todos os cliente que tem empréstimos e a cidade em que vivem.

$$\pi_{\text{cliente-nome, cidade}} (\text{EMPRESTIMO} \; |X| \; \text{CLIENTE})$$

Exemplo

Encontre todos os cliente que tem empréstimos e que moram em “Vitória”.

$$\pi_{\text{cliente-nome, cidade}} (\sigma_{\text{cidade=”Vitória”}} (\text{EMPRESTIMO} \; |X| \; \text{CLIENTE}))$$

Exemplo

Encontre os nomes de todos os clientes que têm conta nas agências situadas em “Vitória”

$$\pi_{\text{cliente-nome, numer-conta}} (\sigma_{\text{agencia-cidade=”Vitória”}} (\text{CLIENTE} \; |X| \; \text{CONTA} \; |X| \; \text{AGENCIA}))$$

2.2.9 Operação Divisão

A operação divisão, representada por ÷, serve para consultas que incluem frases com “para todos”.

Exemplo

Suponha que desejamos encontrar todos os clientes que têm uma conta em todas as agências localizadas em “Vitória”. Podemos obter todas as agências de Vitória através da expressão:

$$r1 \leftarrow \pi_{\text{agencia-cod}} (\sigma_{\text{cidade=”Vitória”}} (\text{AGENCIA}))$$

Podemos encontrar todos os pares cliente-nome, agencia-cod nos quais um cliente possui uma conta em uma agência escrevendo:

$$r2 \leftarrow \pi_{\text{cliente-nome, agencia-cod}} (\text{CONTA} \mid X \mid \text{CLIENTE})$$

Agora precisamos encontrar clientes que apareçam em r2 com cada nome de agência em r1.

Escrevemos esta consulta da seguinte forma:

$$\pi_{\text{cliente-nome, agencia-cod}} (\text{CONTA} \mid X \mid \text{CLIENTE}) \div \pi_{\text{agencia-cod}} (\sigma_{\text{cidade=”Vitória”}} (\text{AGENCIA}))$$

2.3 EXERCÍCIOS

1. Baseado no esquema Banco apresentado, faça as seguintes consultas:

A. Selecionar todos os clientes que possuam conta na mesma cidade onde moram.

$$t \leftarrow \sigma_{\text{CONTA.agencia-cod=Agencias.agencia-cod}} (\text{CONTA X AGENCIA})$$

$$r \leftarrow \pi_{\text{cliente-nome}} (\sigma_{\text{CLIENTE.cidade=agencia.cidade}} (t \text{ X CLIENTE}))$$

B. Encontre todas as agências que possuam clientes com nome “Maria” (conta ou empréstimos).

$$t \leftarrow \sigma_{\text{CONTA.agencia-cod=Agencias.agencia-cod}} (\text{CONTA X AGENCIA})$$

$$t' \leftarrow \sigma_{\text{EMPRESTIMO.agencia-cod= EMPRESTIMO.agencia-cod}} (\text{EMPRESTIMO X AGENCIA})$$

$$r \leftarrow \pi_{\text{cliente-nome}} (\sigma_{\text{CLIENTE.cliente-nome=”Maria” ^ CONTA.cliente-cod = CLIENTE.cliente-cod}} (t \text{ X CLIENTE}))$$

È

$$\pi_{\text{cliente-nome}} (\sigma_{\text{CLIENTE.cliente-nome=”Maria” ^ EMPRESTIMO.cliente-cod = CLIENTE.cliente-cod}} (t' \text{ X CLIENTE}))$$

C. Usando as Operações Fundamentais, encontre a conta com maior saldo.

Relação com os saldos que não são os mais altos:

$$t \leftarrow \pi_{\text{CONTA.saldo}} (\sigma_{\text{CONTA.saldo < d.saldo}} (\text{CONTA X } \rho_d (\text{CONTA}))$$

$$\text{RESPOSTA} \leftarrow \pi_{\text{CONTA.saldo}} (\text{CONTA}) - t$$

2. Baseado no esquema Empresa apresentado abaixo , faça as consultas.

EMPREGADO = (nome, código, endereço, sexo, salário, cod-supervisor, num-departamento)
DEPENDENTE = (cod-empregado, nome, sexo, data-nasc)
DEPARTAMENTO = (nome, número, cod-gerente, data-gerente)
PROJETO = (nome, número, num-dpto, local)
EMPREGADO-PROJETO = (cod-empregado, num-projeto, horas)

A) Encontre o nomes e endereço de todos os empregados que trabalham para o departamento de pesquisa.

B) Para todos os projetos localizados em "Vitória", listar os números dos projetos, os números dos departamentos, e o nome do gerente do departamento.

C) Encontre os empregados que trabalham em todos os projetos controlados pelo departamento número 5.

D) Faça uma lista dos números dos projetos que envolvem um empregado chamado "Chico" como um trabalhador ou como um gerente do departamento que controla o projeto.

E) Liste os nomes dos empregados que não possuem dependentes.

F) Liste os nomes dos gerentes que têm pelo menos um dependente.

2.4 MODIFICANDO O BANCO DE DADOS

As modificações de um banco de dados são expressas em álgebra relacional através do operador de atribuição ←. Veremos as operações de inserção, remoção e alteração (atualização) do banco de dados.

As operações de inserção e remoção, de fato não são novas. Elas serão expostas aqui apenas para fixar o entendimento de suas potencialidades. De fato já temos usado a união e a subtração para incluir e excluir tuplas de relações.

A operação de alteração entretanto traz um conceito novo, ainda não abordado.

2.4.1 Operação de inserção

Esta operação é usada para inserir dados em uma relação.

$$\text{RELAÇÃO1} \leftarrow \text{RELAÇÃO1} \cup \text{RELAÇÃO2}$$

Os valores dos atributos da RELAÇÃO1 e RELAÇÃO2 devem ter domínios equivalentes.

Exemplo

Podemos acrescentar uma nova conta em CONTA da seguinte forma:

$$\text{CONTA} \leftarrow \text{CONTA} \cup \{(51, 980987, 1, 1200)\}$$

2.4.2 Remoção

A remoção é expressa da seguinte forma:

$$\text{RELAÇÃO1} \leftarrow \text{RELAÇÃO1} - \text{RELAÇÃO2}$$

Exemplo

Excluir todas as contas do cliente de código 01

$$\text{CONTA} = \text{CONTA} - (\sigma_{\text{cliente-cod} = \text{“01”}} (\text{CONTA}))$$

Exemplo

Excluir todas as contas de "joão"

$$T \leftarrow (\sigma_{\text{cliente-nome="joão"}} \text{ (CONTA |X| CLIENTE)}$$

$$\text{CONTA} = \text{CONTA} - (\sigma_{\text{cliente-cod = "01"}} \text{ (T)})$$

Exemplo

Suponha que a agência 051 esteja fazendo uma "promoção" e resolva abri uma conta automaticamente para todos os clientes que possuem empréstimo mas não uma conta nesta agência. O número da nova conta será igual ao número do empréstimo (ou de um dos empréstimos caso haja mais de um).

EMPRESTIMO-051 $\leftarrow \sigma_{\text{agencia-cod = 051}}$ (EMPRESTIMO)

CONTA-051 $\leftarrow \sigma_{\text{agencia-cod = 051}}$ (CONTA)

Os clientes que possuem um empréstimo mas não possuem uma conta na agência 051:

CLIENTE-COD $\leftarrow \pi_{\text{cliente-cod}}$(EMPRESTIMO-051 X CLIENTE)

$$- \pi_{\text{cliente-cod}} \text{ (CONTA-051 |X| CLIENTE)}$$

Os novos clientes serão os que não possuem conta, com o número do empréstimo, com um saldo de 0,00 (supomos que o banco seja sério e não quer dar dinheiro a ninguém ☺).

CLIEENTES-NOVOS $\leftarrow (\pi_{\text{cliente-cod,agencia-cod,emprestimo-numero}}$

$$\text{(CLIENTE-COD |X| EMPRESTIMO-051))} \text{ x } \{(0,00)\}$$

A resposta será então:

CONTA $\leftarrow$ CONTA $\cup$ CLIEENTES-NOVOS

2.4.3 Atualização

Em certas situações, podemos desejar mudar um valor em uma tupla sem mudar todos os valores da tupla. Se fizermos estas mudanças usando remoção e inserção podemos não ser capazes de reter os valores que não queremos mudar. Nestes casos usamos o operador atualização, representado pela letra grega δ.

A atualização é uma operação fundamental da álgebra relacional.

$$\delta_{\text{atributo} \leftarrow \text{valor}} \text{ (RELAÇÃO)}$$

onde atributo deve ser um atributo de RELAÇÃO e valor deve ter um domínio compatível.

Exemplo

Suponha que o rendimento das contas tenha sido de 5%. Aumente o saldo de todas as pessoas em 5 %.

$$\delta_{\text{saldo} \leftarrow \text{saldo * 1.05}} \text{ (CONTAS)}$$

Exemplo

Suponhamos que contas com saldos superiores a 10.000 recebam 6% de juros e as demais apenas 5%.

$$\delta_{\text{saldo} \leftarrow \text{saldo * 1.06}} (\sigma_{\text{saldo > 10000}} \text{ (CONTAS))}$$

$$\delta_{\text{saldo} \leftarrow \text{saldo * 1.05}} (\sigma_{\text{saldo <= 10000}} \text{ (CONTAS))}$$

Capítulo 3 SQL – STRUCTURED QUERY LANGUAGE

Quando os Bancos de Dados Relacionais estavam sendo desenvolvidos, foram criadas linguagens destinadas à sua manipulação. O Departamento de Pesquisas da IBM, desenvolveu a SQL como forma de interface para o sistema de BD relacional denominado SYSTEM R, início dos anos 70. Em 1986 o *American National Standard Institute* (ANSI), publicou um padrão SQL. A SQL estabeleceu-se como linguagem padrão de Banco de Dados Relacional.

SQL apresenta uma série de comandos que permitem a definição dos dados, chamada de **DDL (Data Definition Language)**, composta entre outros pelos comandos Create, que é destinado a criação do Banco de Dados, das Tabelas que o compõe, além das relações existentes entre as tabelas. Como exemplo de comandos da classe DDL temos os comandos Create, Alter e Drop.

Os comandos da série **DML (Data Manipulation Language)**, destinados a consultas, inserções, exclusões e alterações em um ou mais registros de uma ou mais tabelas de maneira simultânea. Como exemplo de comandos da classe DML temos os comandos Select, Insert, Update e Delete. Uma subclasse de comandos DML, a **DCL (Data Control Language)**, dispõe de comandos de controle como Grant e Revoke.

A Linguagem SQL tem como grandes virtudes sua capacidade de gerenciar índices, sem a necessidade de controle individualizado de índice corrente, algo muito comum nas linguagens de manipulação de dados do tipo registro a registro (dos modelos Hierárquico e Redes). Outra característica muito importante disponível em SQL é sua capacidade de construção de visões, que são formas de visualizarmos os dados na forma de listagens independente das tabelas e organização lógica dos dados.

Outra característica interessante na linguagem SQL é a capacidade que dispomos de cancelar uma série de atualizações ou de as gravarmos, depois de iniciarmos uma seqüência de atualizações. Os comandos Commit e Rollback são responsáveis por estas facilidades.

Existem inúmeras versões de SQL. A versão original foi desenvolvida no Laboratório de Pesquisa da IBM. Esta linguagem, originalmente chamada Sequel foi implementada como parte do projeto System R no início dos anos 70. A linguagem evoluiu desde então, e seu nome foi mudado para SQL (Structured Query Language).

Em 1986, o *Americam National Standard Institute* (ANSI) publicou um padrão SQL. A IBM publicou o seu próprio SQL standard, o *Systems Aplication Arquitecture Database Interface* (SAA-SQL) em 1987.

Em 1989, tanto ANSI quanto ISO publicaram padrões substitutos (ANSI X3.135-1989 e ISO/IEC 9075:1989) que aumentaram a linguagem e acrescentaram uma capacidade opcional de integridade referencial, permitindo que projetistas de bancos de dados pudessem criar relacionamentos entre dados em diferentes partes do banco de dados. A versão em uso do padrão ANSI / ISSO SQL é o padrão SQL-92 (ANSI X3.135-1992) mas algumas aplicações atuais dão suporte apenas ao padrão SQL-89.

Desde 1993 há um trabalhos sendo desenvolvidos para atualizar o padrão de modo que este atenda às características das últimas versões de bancos de dados relacionais lançadas no mercado. A principal inovação da nova versão (chamada provisoriamente de SQL3) é o suporte à orientação a objetos.

Algumas das características da linguagem SQL são:

- Permite trabalhar com várias tabelas;
- Permite utilizar o resultado de uma instrução SQL em outra instrução SQL (sub-queries);
- Não necessita especificar o método de acesso ao dado;
- É uma linguagem para vários usuários como:
- Administrador do sistema;
- Administrador do banco de dados (DBA);
- Programadores de aplicações;

- Pessoal da agência e tomada de decisão;
- É de fácil aprendizado;
- Permite a utilização dentro de uma linguagem procedural como C, COBOL, FORTRAN, Pascal e PL/I - SQL embutida.

3.1 A ESTRUTURA BÁSICA DE UMA EXPRESSÃO SQL

A estrutura básica de uma expressão SQL consiste em três cláusulas: **select**, **from** e **where**.

- A cláusula **select** corresponde à operação projeção da álgebra relacional. É usada para listar os atributos desejados no resultado de uma consulta.
- A cláusula **from** corresponde à operação produto cartesiano da álgebra relacional. Ela lista as relações a serem examinadas na avaliação da expressão.
- A cláusula **where** corresponde ao predicado de seleção da álgebra relacional. Consiste em um predicado envolvendo atributos de relações que aparecem na cláusula from.

Uma típica consulta SQL segue a seguinte ordem:

select a1, a2, ..., an ← 3ª
from T1, T2, ..., Tm ← 1ª
where P ← 2ª

Cada ai representa um atributo e cada Ti é uma relação e P é um predicado. Esta consulta é equivalente à expressão da álgebra relacional

$$\mathbf{p}_{a1, a2, ..., an} (\mathbf{s}_{P} (T1 \times T2 \times ... \times Tm))$$

A SQL forma o produto cartesiano das relações chamadas na cláusula from, executa uma seleção da álgebra relacional usando o predicado da cláusula where e, então, projeta o resultado para os atributos da cláusula select. Na prática, a SQL pode converter esta expressão em uma forma equivalente que pode ser processada mais eficientemente.

Exemplo

No esquema BANCO, encontre os nomes de todos os clientes de 'Vitória'.

```
select cliente_nome
from CLIENTE
where cidade = 'Vitória'
```

Encontre os nomes de todos os clientes.

```
select cliente_nome
from CLIENTE
```

Exemplo

No esquema EMPRESA, selecionar o nome e o RG dos funcionários que trabalham no departamento número 2 na tabela EMPREGADO

```
select nome, rg
from EMPREGADOS
where depto = 2;
```

obteremos então o seguinte resultado:

Nome	RG
Fernando	20202020

Ricardo	30303030
Jorge	40404040

A consulta acima é originária da seguinte função em álgebra relacional:

$\pi_{nome,\ rg}\ (\sigma_{depto\ =\ 2}\ (EMPREGADOS));$

Em SQL também é permitido o uso de condições múltiplas. Veja o exemplo a seguir:

```
select nome, rg, salario
from EMPREGADOS
where depto = 2 AND salario > 2500.00;
```

que fornece o seguinte resultado:

Nome	**RG**	**Salário**
Jorge	40404040	4.200,00

e que é originária da seguinte função em álgebra relacional:

$\pi_{nome,\ rg,\ salario}\ (\sigma_{depto\ =\ 2\ .and.\ salario\ >\ 3500.00}(EMPREGADOS));$

O operador ***** dentro do especificador *select* seleciona todos os atributos de uma tabela, enquanto que a exclusão do especificador *where* faz com que todas as tuplas de uma tabela sejam selecionadas. Desta forma, a expressão:

```
select *
from empregado;
```

gera o seguinte resultado:

Nome	**RG**	**CIC**	**Depto.**	**RG Supervisor**	**Salário**
João Luiz	10101010	11111111	1	NULO	3.000,00
Fernando	20202020	22222222	2	10101010	2.500,00
Ricardo	30303030	33333333	2	10101010	2.300,00
Jorge	40404040	44444444	2	20202020	4.200,00
Renato	50505050	55555555	3	20202020	1.300,00

3.1.1 Cláusulas Distinct e All

Diferente de álgebra relacional, a operação *select* em SQL permite a geração de tuplas duplicadas como resultado de uma expressão. Para evitar isto, devemos utilizar o cláusula **distinct**.

Exemplo

Sejam as seguintes consultas no esquema EMPRESA.

```
select depto                    select distinct depto
from EMPREGADO;                 from EMPREGADO;
```

que geram respectivamente os seguintes resultados:

Depto.
1

Depto.
1

2
2
2
3

2
3

A cláusula All é o default para o select ou seja: Select All indica para obter todas as tuplas. Logo, esta cláusula não precisa ser colocado (a não ser, talvez por motivos de documentação).

3.1.2 Predicados e ligações

A SQL não têm uma representação da operação ligação natural. No entanto, uma vez que a ligação natural é definida em termos de um produto cartesiano, uma seleção e uma projeção, é relativamente simples escrever uma expressão SQL para uma ligação natural.

Exemplo

Encontre os nomes e cidades de clientes que possuam empréstimos em alguma agência.

```
Select distinct cliente_nome, cidade
From Cliente, Emprestimo
Where Cliente.cliente_cod=Emprestimo.cliente_cod
```

A SQL inclui os conectores **and**, **or** e **not**; caracteres especiais: (,), ., :, _, %<, >, <= , >= , = , <>, +, - ,* e /; operador para comparação: **between**, como mostra o exemplo a seguir.

Exemplo

Selecionar todas as contas que possuam saldo entre 10000 e 20000.

```
Select conta_numero
From CONTA
Where saldo >= 10000 and saldo <= 20000
```

Que equivale à consulta

```
Select conta_numero
From CONTA
Where saldo between 10000 and 20000
```

A SQL inclui também um operador para comparações de cadeias de caracteres, o **like**. Ele é usado em conjunto com dois caracteres especiais:

- Por cento (**%**). Substitui qualquer subcadeia de caracteres.
- Sublinhado (_). Substitui qualquer caractere.

Exemplo

Encontre os nomes de todos os clientes cujas ruas incluem a subcadeia ‘na’

```
Select distinct cliente_nome
From CLIENTE
Where rua like ‘%na%’
```

Exemplo

Encontre os nomes de todos os clientes cujas ruas finalizem com a subcadeia ‘na’, seguido de um caractere.

```
Select distinct cliente_nome
From CLIENTE
Where rua like ‘%na_’
```

Para que o padrão possa incluir os caracteres especiais (isto é, % , _ , etc...), a SQL permite a especificação de um caractere de escape. O caractere de escape é usado imediatamente antes de um caractere especial para indicar que o caractere especial deverá ser tratado como um caractere normal. Definimos o caractere de escape para uma comparação **like** usando a palavra-chave **escape**. Para ilustrar, considere os padrões seguintes que utilizam uma barra invertida como caractere de escape.

- Like ‘ ab\%cd%’ escape ‘\’ substitui todas as cadeias começando com ‘ ab%cd’;
- Like ‘ ab_cd%’ escape ‘\’ substitui todas as cadeias começando com ‘ ab_cd’.

A procura por não-substituições em vez de substituições dá-se através do operador **not like**.

3.2 VARIÁVEIS TUPLAS (RENOMEAÇÃO)

Exemplo

No esquema EMPRESA, selecione o número do departamento que controla projetos localizados em Rio Claro;

```
select t1.numero_depto
from departamento_projeto as t1, projeto as t2
where t1.numero_projeto = t2.numero;
```

Na expressão SQL acima, *t1* e *t2* são chamados *“alias”* (apelidos) e representam a mesma tabela a qual estão referenciando. Um “alias” é muito importante quando há redundância nos nomes das colunas de duas ou mais tabelas que estão envolvidas em uma expressão. Ao invés de utilizar o “alias”, é possível utilizar o nome da tabela, mas isto pode ficar cansativo em consultas muito complexas além do que, impossibilitaria a utilização da mesma tabela mais que uma vez em uma expressão SQL. A palavra chave **as** é opcional.

Exemplo

No esquema EMPRESA, selecione selecione o nome e o RG de todos os funcionários que são supervisores.

```
select e1.nome as “Nome do Supervisor”, e1.rg as “RG do Supervisor”
from empregado e1, empregado e2
where e1.rg = e2.rg_supervisor;
```

que gera o seguinte resultado:

Nome do Supervisor	RG do Supervisor
João Luiz	10101010
Fernando	20202020

Observação: A consulta acima é decorrente da seguinte expressão em álgebra relacional:

$\pi_{nome,\ rg}$ (EMPREGADOS |X| $_{tg\_t1\ =\ rg\_supervisor\_t2}$ EMPREGADOS)

Exemplo

Encontre o nome e a cidade de todos os clientes com uma conta em qualquer agência.

```
Select distinct C.cliente_nome, C.cidade
from CLIENTE C, CONTA S
where C.cliente_cod = S.cliente_cod
```

3.3 OPERAÇÕES DE CONJUNTOS

A SQL inclui as operações de conjunto **union**, **intersect** e **minus** que operam em relações e correspondem às operações **È**, **Ç** e **–** da álgebra relacional.

Uma vez que as relações são conjuntos, na união destas, as linhas duplicadas são eliminadas. Para que uma operação R ∪ S, R ∩ S ou R - S seja válida, necessitamos que duas condições sejam cumpridas:

- As relações R e S devem ter o mesmo número de atributos;
- Os domínios do i-ésimo atributo de R e do i-ésimo atributo de S devem ser os mesmos.

Observação: Nem todos os interpretadores SQL suportam todas as operações de conjunto. Embora a operação union seja relativamente comum, são raros os que suportam intersect ou minus.

Exemplo

Mostrar o nome dos clientes que possuem conta, empréstimo ou ambos na agência de código ‘ 051’:

Empréstimo na agência ‘ 051’:

```
Select distinct cliente_nome
From CLIENTE, EMPRESTIMO
Where CLIENTE.cliente_cod= EMPRESTIMO.cliente_cod and
      agencia_cod = ‘051’
```

Conta na agência ‘ 051’:

```
Select distinct cliente_nome
From  CLIENTE, CONTA
Where CLIENTE.cliente_cod= CONTA.cliente_cod and
      CONTA.agencia_cod = ‘051’
```

Fazendo a união dos dois:

```
(Select distinct cliente_nome
 From CLIENTE, EMPRESTIMO
 Where CLIENTE.cliente_cod= EMPRESTIMO.cliente_cod and
       agencia_cod = ‘051’ )
Union
(Select distinct cliente_nome
 From CLIENTE, CONTA
 Where CLIENTE.cliente_cod= CONTA.cliente_cod and
       CONTA.agencia_cod = ‘051’ )
```

Exemplo

Achar todos os clientes que possuam uma conta e um empréstimo na agência de código ‘ 051’.

```
(Select distinct cliente_nome
 From CLIENTE, EMPRESTIMO
 Where CLIENTE.cliente_cod= EMPRESTIMO.cliente_cod and
       agencia_cod = ‘051’ )
intersect
(Select distinct cliente_nome
 From CLIENTE, CONTA
 Where CLIENTE.cliente_cod= CONTA.cliente_cod and
       CONTA.agencia_cod = ‘051’ )
```

Exemplo

Achar todos os clientes que possuem uma conta mas não possuem um empréstimo na agência de código ‘ 051’.

```
(Select distinct cliente_nome
 From CLIENTE, EMPRESTIMO
```

```
 Where CLIENTE.cliente_cod=Emprestimos.cliente_cod and
       EMPRESTIMO.agencia_cod = '051' )
minus
(Select distinct cliente_nome
 From CLIENTE, CONTA
 Where CLIENTE.cliente_cod= CONTA.cliente_cod and
       CONTA.agencia_cod = '051' )
```

3.4 EXERCÍCIOS

Baseado no esquema EMPRESA, faça as seguintes consultas SQL.

1. Selecione todos os empregados com salário maior ou igual a 2000,00.
2. Selecione todos os empregados cujo nome começa com 'J'.
3. Mostre todos os empregados que têm 'Luiz' ou 'Luis' no nome.
4. Mostre todos os empregados do departamento de 'Engenharia Civil'.
5. Mostre todos os nomes dos departamentos envolvidos com o projeto 'Motor 3'.
6. Liste o nome dos empregados envolvidos com o projeto 'Financeiro 1'.
7. Mostre os funcionários cujo supervisor ganha entre 2000 e 2500.

Baseado no esquema BANCO, faça as seguintes consultas SQL.

8. Liste todas as cidades onde há agências.
9. Liste todas as agências existentes em 'Vitória'.
10. Liste todas as agências que possuem conta com saldo maior que 100.000,00.
11. Mostre os dados de todos os clientes da agência 'Centro' da cidade de 'Vitória' que têm saldo negativo.
12. Liste os dados dos clientes que possuem conta mas não possuem empréstimo na agência de código '005'.

3.5 ORDENANDO A EXIBIÇÃO DE TUPLAS (ORDER BY)

A cláusula **order by** ocasiona o aparecimento de tuplas no resultado de uma consulta em uma ordem determinada. Para listar em ordem alfabética todos os clientes do banco, fazemos:

```
Select distinct cliente_nome
From CLIENTE
Order by cliente_nome
```

Como padrão, SQL lista tuplas na ordem ascendente. Para especificar a ordem de classificação, podemos especificar **asc** para ordem ascendente e **desc** para descendente. Podemos ordenar uma relação por mais de um elemento. Como se segue:

```
Select *
From EMPRESTIMO
Order by quantia desc, agencia_cod asc
```

Para colocar as tuplas (linhas) em ordem é realizada uma operação de *sort*. Esta operação é relativamente custosa e portanto só deve ser usada quando realmente necessário.

3.6 MEMBROS DE CONJUNTOS

O conectivo **in** testa os membros de conjunto, onde o conjunto é uma coleção de valores produzidos por uma cláusula select. Da mesma forma, pode ser usada a expressão **not in**.

Exemplo

Selecione todas as agências com código 1, 2 ou 3.

```
select *
from agencia
where agencia_cod in (1,2,3)
```

Exemplo

Selecione o nome de todos os funcionários que trabalham em projetos localizados em Rio Claro.

```
select e1.nome, e1.rg, e1.depto
from empregado e1, empregado_projeto e2
where e1.rg = e2.rg_empregado and
      e2.numero_projeto in
      (select numero
       from projeto
       where localizacao = 'Rio Claro');
```

Exemplo

Encontre todos os clientes que possuem uma conta e um empréstimo na agência 'Princesa Isabel'.

```
Select distinct cliente_nome
From CLIENTE
Where CLIENTE.cliente_cod in

      (select cliente_cod
       from CONTA, AGENCIA
```

```
        where CONTA.agencia_cod = AGENCIA.agencia_cod and
              agencia_nome = 'Princesa Isabel')

      and CLIENTE.cliente_cod in

      (select cliente_cod
       from EMPRESTIMO, AGENCIA
       where EMPRESTIMO.agencia_cod= AGENCIA.agencia_cod and
             agencia_nome = 'Princesa Isabel')
```

Exemplo

Encontre todas as agências que possuem ativos maiores que alguma agência de Vitória.

```
select distinct t.agencia_nome
from AGENCIA t, AGENCIA s
where t.ativos > s.ativos and s.cidade = 'Vitória'
```

Observação: Uma vez que isto é uma comparação "maior que", não podemos escrever a expressão usando a construção **in**.

A SQL oferece o operador **some** (equivalente ao operador **any**), usado para construir a consulta anterior. São aceitos pela linguagem: **>some, <some, >=some, <=some, =some**

```
select agencia_nome
from AGENCIA
where ativos > some
      (select ativos
       from AGENCIA
       where agencia_cidade = 'Vitória')
```

Como o operador **some**, o operador **all** pode ser usado como: **>all, <all, >=all, <=all, =all e <>all**. A construção **> all** corresponde a frase "maior que todos".

Exemplo

Encontrar as agências que possuem ativos maiores do que todas as agências de Vitória.

```
select agencia_nome
from AGENCIA
where ativos > all
      (select ativos
       from AGENCIA
       where agencia_cidade = 'Vitória')
```

A SQL possui um recurso para testar se uma subconsulta tem alguma tupla em seus resultados. A construção **exists** retorna **true** se o argumento subconsulta está não-vazio. Podemos usar também a expressão **not exists**.

Exemplo

No esquema EMPRESA, lise o nome dos gerentes que têm ao menos um dependente.

```
Select nome
from EMPREGADO
where exists
      (select *
       from DEPENDENTES
            where DEPENDENTES.rg_responsavel = EMPREGADO.rg)
      and exists
      (select *
       from DEPARTAMENTO
            where DEPARTAMENTO.rg_gerente = EMPREGADO.rg)
```

Exemplo

Usando a construção **exists**, encontre todos os clientes que possuem uma conta e um empréstimo na agência 'Princesa Isabel'.

```
Select cliente_nome
from CLIENTE
where exists
      (select *
       from CONTA, AGENCIA
       where CONTA.cliente_cod= CLIENTE.cliente_cod and
             AGENCIA.agencia_cod = CONTA.agencia_cod and
             agencia_nome = 'Princesa Isabel')
      and exists
      (select *
       from EMPRESTIMO, AGENCIA
       where EMPRESTIMO.cliente_cod= CLIENTE.cliente_cod and
             AGENCIA.agencia_cod = EMPRESTIMO.agencia_cod and
             agencia_nome = 'Princesa Isabel')
```

3.7 FUNÇÕES AGREGADAS

A SQL oferece a habilidade para computar funções em grupos de tuplas usando a cláusula **group by**. O(s) atributo(s) dados na cláusula group by são usados para formar grupos. Tuplas com o mesmo valor em todos os atributos na cláusula group by são colocados em um grupo. A SQL inclui funções para computar:

Média: **avg** Mínimo: **min** Máximo: **max** Soma: **sum** Contar: **count**

Exemplo

Encontre o saldo médio de conta em cada agência.

```
Select agencia_nome, avg(saldo)
From CONTA, AGENCIA
where CONTA.agencia_cod = AGENCIA.agencia_cod
Group by agencia_nome
```

Exemplo

Encontre o número de depositantes de cada agência.

```
Select agencia_nome, count(distinct cliente_nome)
From CONTA, AGENCIA
where CONTA.agencia_cod = AGENCIA.agencia_cod
Group by agencia_nome
```

Observação: Note que nesta última consulta é importante a existência da cláusula distinct pois um cliente pode Ter mais de uma conta em uma agência, e deverá ser contado uma única vez.

Exemplo

Encontre o maior saldo de cada agência.

```
Select agencia_nome, max(saldo)
From CONTA, AGENCIA
Where CONTA.agencia_cod= AGENCIA.agencias_cod
Group by agencia_nome
```

Ás vezes, é útil definir uma condição que se aplique a grupos em vez de tuplas. Por exemplo, poderíamos estar interessados apenas em agências nas quais a média dos saldos é maior que 1200. Esta condição será aplicada a cada grupo e não à tuplas simples e é definida através da cláusula **having**. Expressamos esta consulta em SQL assim:

```
Select agencia_nome, avg(saldo)
From CONTA, AGENCIA
Where CONTA.agencia_cod= AGENCIA.agencias_cod
Group by agencia_nome Having avg(saldo)>1200
```

Às vezes, desejamos tratar a relação inteira como um grupo simples. Nesses casos, não usamos a cláusula **group by**.

Exemplo

Encontre a média de saldos de todas as contas.

```
Select avg(saldo)
From CONTA
```

3.8 MODIFICANDO O BANCO DE DADOS

3.8.1 Remoção

Podemos remover somente tuplas inteiras, não podemos remover valores apenas em atributos particulares.

Sintaxe:

```
Delete R
Where P
```

onde R representa uma relação e P um predicado.

Note que o comando **delete** opera em apenas uma relação. O predicado da cláusula **where** pode ser tão complexo como o predicado **where** do comando **select**.

Exemplo

Remover todas as tuplas de empréstimo.

```
delete EMPRESTIMO
```

Exemplo

Remover todos os registros da conta de 'João'.

```
delete CONTA
where cliente_cod in
      (select cliente_cod
       from CLIENTE
       where cliente_nome = 'João')
```

Exemplo

Remover todos os empréstimos com números entre 1300 e 1500.

```
delete EMPRESTIMO
where emprestimo_numero between 1300 and 1500
```

Exemplo

Remova todas as contas de agências localizadas em 'Vitória'.

```
Delete CONTA
where agencia_cod in
      (select agencia_cod
       from AGENCIA
       where agencia_cidade='Vitoria')
```

3.8.2 Inserção

Para inserir um dado em uma relação, ou especificamos uma tupla para ser inserida escrevemos uma consulta cujo resultado seja um conjunto de tuplas a serem inseridas. Os valores dos atributos para tuplas inseridas precisam necessariamente ser membros do mesmo domínio do atributo.

Exemplo

Inserir uma nova conta para João (código = 1), número 9000, na agência de código=2 cujo valor seja 1200.

```
insert into CONTA
values (2,9000,1,1200)
```

Na inserção acima é considerando a ordem na qual os atributos correspondentes estão listados no esquema relação. Caso o usuário não se lembre da ordem destes atributos, pode fazer o mesmo comando da seguinte forma:

```
insert into CONTA (agencia_cod, conta_numero, cliente_cod, saldo)
values (2,9000,1,1200)
```

Podemos querer também inserir tuplas baseadas no resultado de uma consulta.

Exemplo

Inserir todos os clientes que possuam empréstimos na agência 'Princesa Isabel' na relação CONTA com um saldo de 200. O número da nova conta é o número do empréstimo * 10000.

```
insert into CONTA (agencia_cod, conta_numero, cliente_cod, saldo)
select AGENCIA.agencia_cod, emprestimo_numero*10000, cliente_cod, 200
from EMPRESTIMO, AGENCIA
where EMPRESTIMO.agencia_cod= AGENCIA.agencia_cod and
      agencia_nome = 'Princesa Isabel'
```

3.8.3 Atualizações

Em certas situações, podemos desejar mudar um valor em uma tupla sem mudar todos os valores na tupla. Para isso, o comando **update** pode ser usado.

Suponha que esteja sendo feito o pagamento de juros, e que em todos saldos sejam acrescentados em 5%. Escrevemos

```
update CONTA
set saldo = saldo * 1,05
```

Suponha que todas as contas com saldo superiores a 10000 recebam aumento de 6% e as demais de 5%.

```
Update CONTA
set saldo = saldo * 1,06
where saldo >10000

Update CONTA
set saldo = saldo * 1,05
where saldo<=10000
```

A cláusula **where** pode conter uma série de comandos select aninhados. Considere, por exemplo, que todas as contas de pessoas que possuem empréstimos no banco terão acréscimo de 1%.

```
Update CONTA
set saldo = saldo * 1,01
where cliente_cod in
      (select cliente_cod
       from EMPRESTIMO )
```

Exemplo

No esquema EMPRESA, atualize o salário de todos os empregados que trabalham no departamento 2 para R$ 3.000,00.

```
update empregado
set    salario = 3000
where  depto = 2;
```

Exemplo

No esquema BANCO, atualize o valor dos ativos. Os ativos são os valores dos saldos das contas da agência.

```
update agencia
set ativos =
      (select sum(saldo)
       from conta
       where conta.agencia_cod = agencia.agencia_cod)
```

3.8.4 Valores Nulos

É possível dar valores a apenas alguns atributos do esquema para tuplas inseridas em uma dada relação. Os atributos restantes são designados como nulos. Considere a requisição:

```
insert into CLIENTE (cliente_cod, cliente_nome, rua, cidade)
values (123,'Andrea',null,null)
```

A palavra chave **null** pode ser usada em um predicado para testar se um valor é nulo. Assim, para achar todos os clientes que possuem valores nulos para rua, escrevemos:

```
select distinct cliente_nome
from CLIENTE
where rua is null
```

O predicado **is not null** testa a ausência de um valor nulo.

3.9 DEFINIÇÃO DE DADOS

O conjunto de relações de um Banco de Dados precisa ser especificado ao sistema por meio de uma linguagem de definição de dados - DDL. A SQL DDL permite a especificação não apenas de um conjunto de relações, mas também de informações sobre cada relação, incluindo:

- Esquema para cada relação;
- Domínio de valores associados a cada atributo;
- Conjunto de índices a ser mantido para cada relação;
- Restrições de integridade;
- A estrutura física de armazenamento de cada relação no disco.

Uma relação SQL é definida usando o comando **create table**. A forma geral do comando **create table** então é:

```
create table <nome_tabela> (
      <nome_coluna1> <tipo_coluna1> <NOT NULL>,
      <nome_coluna2> <tipo_coluna2> <NOT NULL>,
      ...
      <nome_colunan> <tipo_colunan> <NOT NULL>);
```

A restrição **not null** indica que o atributo deve ser obrigatoriamente preenchido; se não for especificado, então o *default* é que o atributo possa assumir o valor nulo.

Exemplo

Criar a tabela EMPREGADO do esquema EMPRESA, teríamos o seguinte comando:

```
create table EMPREGADO
      (nome  char (30) NOT NULL,
       rg    integer   NOT NULL,
       cic   integer,
       depto integer   NOT NULL,
       rg_supervisor   integer,
       salario, decimal (7,2)NOT NULL)
```

O comando **create table** inclui opções para especificar certas restrições de integridade, conforme veremos.

A relação criada acima está inicialmente vazia. O comando insert poderá ser usado para carregar os dados para uma relação.

Para remover uma tabela de banco de dados SQL, usamos o comando **drop table**. O comando **drop table** remove todas as informações sobre a relação retirada do banco de dados. A forma geral para o comando **drop table** é:

```
drop table <nome_tabela>;
```

Exemplo

Eliminar a tabela EMPREGADO do esquema EMPRESA, teríamos o seguinte comando:

```
drop table EMPREGADOS;
```

Observe que neste caso, a chave da tabela EMPREGADOS, (rg) é utilizada como chave estrangeira ou como chave primária composta em diversos tabelas que devem ser devidamente corrigidas. Este processo não é assim tão simples pois, como vemos neste caso, a exclusão da tabela EMPREGADO implica na alteração do projeto físico de diversas tabelas. Isto acaba implicando na construção de uma nova base de dados.

O comando **alter table** é usado para alterar a estrutura de uma relação existente. Permite que o usuário faça a inclusão de novos atributos em uma tabela. A forma geral para o comando **alter table** é a seguinte:

```
alter table <tabela> <add,drop,modify> <coluna> <tipo_coluna>;
```

onde *add*, adiciona uma coluna; *drop*, remove uma coluna; e *modify*, modifica algo em uma tabela.

No caso do comando **alter table**, a restrição NOT NULL não é permitida pois assim que se insere um novo atributo na tabela, o valor para o mesmo em todas as tuplas da tabela receberão o valor NULL.

Não é permitido eliminar algum atributo de uma relação já definida. Assim caso você desejar eliminar uma chave primária devidamente referenciada em outra tabela como chave estrangeira, ao invés de obter a eliminação do campo, obterá apenas um erro.

3.10 VISÕES (VIEWS)

Uma view em SQL é uma tabela que é derivada de outras tabelas ou de outras views. Uma view não necessariamente existe em forma física; é considerada uma tabela virtual (em contraste com as tabelas cujas tuplas são efetivamente armazenadas no banco de dados).

Uma view é definida usando o comando **create view**. Para definir uma visão, precisamos dar a ela um nome e definir a consulta que a processa.

A forma geral é:

```
create view <nomevisao> as <expressão de consulta>
```

onde, <expressão de consulta> é qualquer consulta SQL válida.

Como exemplo, considere a visão consistindo em nomes de agências e de clientes.

```
Create view TOTOS_CLIENTES as
(select agencia_nome,cliente_nome
 from CLIENTE, CONTA, AGENCIA
 where CLIENTE.cliente_cod = CONTA.cliente_cod and
       CONTA.agencia_cod = AGENCIA.agencia_cod)
union

(select agencia_nome,cliente_nome
 from CLIENTE, EMPRESTIMO, AGENCIA
 where CLIENTE.cliente_cod = EMPRESTIMO.cliente_cod and
       EMPRESTIMO.agencia_cod = AGENCIA.agencia_cod)
```

Nomes de visões podem aparecer em qualquer lugar onde nome de relação (tabela) possa aparecer. Usando a visão TOTOS_CLIENTES, podemos achar todos os clientes da agência 'Princesa Isabel', escrevendo:

```
select cliente_nome
from TOTOS_CLIENTES
where agencia_nome = 'Princesa Isabel'
```

Uma modificação é permitida através de uma visão apenas se a visão em questão está definida em termos de uma relação do atual banco de dados relacional.

Exemplo

Selecionando tuplas de empréstimos.

```
Create view emprestimo_info as
(select agencia_cod, emprestimo_numero, cliente_cod
 from EMPRESTIMO)
```

Uma vez que a SQL permite a um nome de visão aparecer em qualquer lugar em que um nome de ralação aparece, podemos escrever:

```
insert into emprestimo_info
values (1,40,7)
```

Esta inserção é representada por uma inserção na relação EMPRESTIMO, uma vez que é a relação a partir do qual a visão emprestimo_info foi construída. Devemos, entretanto, ter algum valor para *quantia*. Este valor é um valor nulo. Assim, o **insert** acima resulta na inserção da tupla: (1,40,7,null) na relação EMPRESTIMO.

Da mesma forma, poderíamos usar os comandos *update,* e *delete*.

Para apagar uma visão, usamos o comando

```
drop view <nomevisão>
```

Exemplo

Apagar a visão emprestimo_info.

```
drop view emprestimo_info
```

3.11 RESTRIÇÕES DE INTEGRIDADE

As restrições de integridade fornecem meios para assegurar que mudanças feitas no banco de dados por usuários autorizados não resultem na inconsistência dos dados.

Há vários tipos de restrições de integridade que seriam cabíveis a bancos de dados. Entretanto as regras de integridade são limitadas às que podem ser verificadas com um mínimo de tempo de processamento.

3.11.1 Restrições de domínios

Um valor de domínio pode ser associado a qualquer atributo. Restrições de domínio são as mais elementares formas de restrições de integridade. Elas são facilmente verificadas pelo sistema sempre que um novo item de dado é incorporado ao banco de dados.

O princípio que está por traz do domínio de atributos é similar aos dos tipos em linguagem de programação. De fato é possível criar domínios em SQL através do comando **create domain**.

A forma geral é:

```
CREATE DOMAIN nome_do_domínio tipo_original
[ [ NOT ] NULL ]
[ DEFAULT valor_default ]
[ CHECK ( condições ) ]
```

Exemplo

Cria o tipo_codigo como um número de 3 algarismos com valores permitidos de 100 a 800.

```
create domain tipo_codigo numeric(3)
not null
check ((tipo_codigo >= 100) and (tipo_codigo <= 800))
```

A cláusula **check** da SQL-92 permite modos poderosos de restrições de domínio. Ela pode ser usada também no momento de criação da tabela.

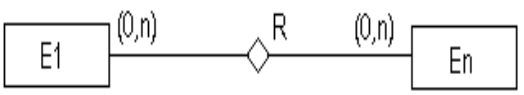
Exemplo

```
create table dept
(deptno integer(2) not null,
 dname char(12),
 loc char(12),
 check (dptono >= 100))
```

3.11.2 Restrições de integridade referencial

Muitas vezes, desejamos assegurar que um valor que aparece em uma relação para um dado conjunto de atributos, apareça também para um certo conjunto de atributos em outra relação. Isto é chamado de *Integridade Referencial*.

Neste relacionamento, considerando **K1** como sendo *chave primária* de **E1** e **Kn** como sendo *chave primária* de **En**, temos em **R** tuplas que serão identificadas inclusive por **K1** e **Kn** (na forma de chaves estrangeiras). Desta forma não poderá



existir um elemento **K1** em **R** que não faça parte de **E1**, tão pouco um **Kn** em **R** que não faça parte de **En.**

As modificações no banco de dados podem causar violações de integridade referencial. Considerações devem ser feitas ao inserir, remover e atualizar tuplas.

Considerar: $\pi_{\alpha}(R2) \subseteq \pi_{k}(R1)$

- **Inserir**. Se uma tupla t2 é inserida em R2, o sistema precisa assegurar que existe uma tupla t1 em R1 tal que $t_1[k] = t_2[\alpha]$. Isto é, $t_2[\alpha] \varepsilon \pi_k(R1)$
- **Remover**. Se uma tupla t2 é removida de R2, o sistema precisa computar o conjunto de tuplas em R2que referencia t1:

  $$\sigma_{\alpha}(R1) = t_{1[k]}(R2)$$

  Se este conjunto não for vazio, o comando remover é rejeitado como um erro, ou as tuplas que se referem a t1 precisam elas mesmas ser removidas. A última solução pode levar a uma remoção em cascata, uma vez que as tuplas precisam referir-se a tuplas que se referem a t1 e assim por diante.
- **Atualizar**. Precisamos considerar dois casos para atualização: a atualização da relação referenciadora (filha - R2) e atualização da relação referenciada (pai - R1).

  Se a tupla t2 é atualizada na relação R2 e a atualização modifica valores para a chave estrangeira α, então é feito um teste similar para o caso de inserção. Digamos que t2 denote o novo valor da tupla t2'. O sistema precisa assegurar que:

  $$t_{2'}[\alpha] \in \pi_{k}(R1)$$

  Se a tupla t1 é atualizada em R1, e a atualização modifica valores para a chave primária (K), então é feito um teste similar ao caso remover. O sistema precisa computar

  $$\sigma_{\alpha}(R1) = t_{1[k]}(R2)$$

A SQL permite a especificação de chaves primárias, candidatas e estrangeiras como parte da instrução **create table**.

- A cláusula **primary key** da instrução *create table* inclui uma lista de atributos que compreende a *chave primária*;
- A cláusula **unique key** da instrução *create table* inclui uma lista de atributos que compreende a *chave candidata*;
- A cláusula **foreign key** da instrução *create table* inclui uma lista de atributos que compreende a *chave estrangeira* e o nome da relação referida pela chave estrangeira.

Exemplo

Criar as relações cliente, agencia e conta para o esquema do banco.

```
Create table CLIENTE
(cliente_cod integer not null,
 cliente_nome char(30),
 rua char(30),
 cidade char(30),
 primary key (cliente_cod))

Create table AGENCIA
(agencia_cod integer not null,
 agencia_nome char(30),
 agencia_cidade char(30),
 fundos decimal (7,2),
```

```
primary key (agencia_cod),
check (fundos >= 0))
```

```
Create table  CONTA
(agencia_cod int,
 conta_numero char(10) not null,
 cliente_cod int not null,
 saldo decimal (7,2),
 primary key (cliente_cod,conta_numero),
 foreign key (cliente_cod) references clientes,
 foreign key (agencia_cod) references agencias,
 check (fundos >= 0))
```

Notem que os atributos chaves precisam ter a especificação de **not null**.

Quando uma regra de integridade referencial é violada, o procedimento normal é rejeitar a ação que ocasionou essa violação. Entretanto é possível criar ações para modificação das tabelas associadas onde houve (ou haveria) a quebra de integridade referencial.

Isto é feito através das cláusulas **on delete cascade** e **on update cascade** associadas à clausula **foreign key**.

Exemplo

```
Create table  CONTA
(...

 foreign key (agencia_cod) references agencia)
 on delete cascade on update cascade,

 ...)
```

Neste exemplo, se a remoção de uma tupla da tabela agência resultar na violação da regra de integridade, o problema é resolvido removendo as tuplas associadas com esta agência da tabela conta. De forma semelhante, se o código de uma agência for alterado, as contas desta agência serão também alteradas.

3.11.3 Asserções

Uma asserção (afirmação) é um predicado expressando uma condição que desejamos que o banco de dados sempre satisfaça. Quando uma assertiva é criada, o sistema testa sua validade. Se a afirmação é válida, então qualquer modificação posterior no banco de dados será permitida apenas quando a asserção não for violada.

Restrições de domínio e regras de integridade referencial são formas especiais de asserções.

O alto custo de testar e manter afirmações tem levado a maioria de desenvolvedores de sistemas a omitir o suporte para afirmações gerais. A proposta geral para a linguagem SQL inclui uma construção de proposta geral chamada instrução **create assertion** para a expressão de restrição de integridade. Uma afirmação pertencente a uma única relação toma a forma:

```
create assertion <nome da asserção> check <predicado>
```

Exemplo

Definir uma restrição de integridade que não permita saldos negativos.

```
create assert saldo_restricao
check (not exists (select * from CONTA where saldo < 0) )
```

Exemplo

Só permitir a inserção de saldos maiores que quantias emprestadas para aquele cliente.

```
create assert saldo_restricao2
check (not exists
(select *
 from conta
```

```
where saldo <
      (select max(quantia)
       from EMPRESTIMO
       where EMPRESTIMO.cliente_cod = CONTA.cliente_cod))
```

3.12 EXECÍCIOS

Observação. *Os seguintes exercícios, bem como a maior parte do texto destas notas de aula, foram tirados das notas de aula de Bancos de Dados 98/01 da Professora Claudinete Vicente Borges.*

Considerando o esquema:

```
Pessoas  = (CodPessoa, NomePessoa, Cidade, Chefe)
Empresas = (CodEmpresa, NomeEmpresa, Cidade)
Trabalha = (CodPessoa, CodEmpresa, Salario)
```

1. **Consulte todas as pessoas que trabalham em Vitória.**

```
Select Pessoas.NomePessoa
from Pessoas
where Cidade = 'Vitória'
```

2. **Consulte todas as pessoas que trabalham na mesma cidade onde moram.**

```
Select Pessoas.NomePessoa
from Pessoas, Trabalha, Empresas
where Pessoas.CodPessoa = Trabalha.CodPessoa and
      Trabalha.CodEmpresa = Empresas.CodEmpresa and
      Pessoa.Cidade = Empresas.Cidade
```

3. **Consulte todas as pessoas que moram na mesma cidade do chefe.**

```
Select Pessoas.NomePessoa
from Pessoas, Pessoas Chefes
where Pessoas.Chefe = Chefes.CodPessoa and
      Pessoa.Cidade = Chefes.Cidade
```

4. **Consulte todas as empresas que funcionam em cidades que não moram pessoas cujo primeiro nome seja Maria (usar operações de conjunto).**

```
Select Empresas.NomeEmpresa
from Empresas
Where Empresas.Cidade not in
      (select Cidade
       from Pessoas
       Where Pessoas.NomePessoa like 'Maria%')
```

5. **Consulte todas as pessoas que não trabalham em Vitória e que ganham acima de 2000 em ordem decrescente da cidade onde moram.**

```
Select Pessoas.NomePessoa
from Pessoas, Trabalha, Empresas
where Pessoas.CodPessoa = Trabalha.CodPessoa and
      Trabalha.CodEmpresa = Empresas.CodEmpresa and
      Empresas.Cidade < > 'Vitória' and Salario > 2000
order by pessoas.cidade desc
```

6. **Consulte todas as pessoas que não trabalham na empresa que comece com 'inf_' em ordem alfabética.**

```
Select Pessoas.NomePessoa
from Pessoas, Trabalha, Empresas
where Pessoas.CodPessoa = Trabalha.CodPessoa and
      Trabalha.CodEmpresa = Empresas.CodEmpresa and
      Empresas.nomeEmpresa not like 'inf&_%' scape '&'
order by nomeEmpresa
```

Considere o esquema seguinte para as questões que se seguem.

```
Fabricante  =(codf, nomef)
Automovel   =(coda, nomea, preço, codf)
Pessoa      =(codp, nomep)
Venda       =(codp, coda, valor, cor, data)
```

7. Quem comprou 'Ford'? (fazer duas consultas).

```
select Pessoa.NomeP
from Pessoa, venda
where Pessoa.codp = Venda.codp and
      venda.coda in
            (select coda
             from  Automovel, Fabricante
             where Automovel.codf = Fabricante.codf
             and Fabricante.Nomef = 'Ford')
```

Outra resposta:

```
select Pessoa.NomeP
from Pessoa, venda, Automovel, Fabricante
where Pessoa.codp = Venda.codp and
      venda.coda = Automovel.coda and
      Automovel.codf = Fabricante.codf and
      Fabricante.Nomef = 'Ford'
```

8. Quem não comprou 'Ford'?

```
select Pessoa.NomeP
from Pessoa, venda
where Pessoa.codp = Venda.codp and
      venda.coda not in
      (select coda
       from  Automovel, Fabricante
       where Automovel.codf = Fabricante.codf and
             Fabricante.Nomef = 'Ford')
```

9. Quem comprou carro com ágio?

```
select Pessoa.NomeP
from Pessoa, venda, Automovel
where Pessoa.codp = Venda.codp and
      venda.coda = Automovel.coda and
      venda.valor > Automovel.preço
```

10. Quem comprou 'Ford' e não comprou 'Volks'?

```
select Pessoa.NomeP
from Pessoa, venda
where Pessoa.codp = Venda.codp

      and venda.coda in
```

```
        (select coda
         from  Automovel, Fabricante
         where Automovel.codf = Fabricante.codf
         and Fabricante.Nomef = 'Ford')

and venda.coda not in

        (select coda
         from  Automovel, Fabricante
         where Automovel.codf = Fabricante.codf
         and Fabricante.Nomef = 'Volks')
```

11. Quem comprou carro entre 01/01/97 e 01/01/98?

```
select Pessoa.NomeP
from Pessoa, venda
where Pessoa.codp = Venda.codp and
      venda.data between '01/01/97' and '01/01/98'
```

Considere o esquema BANCO, reproduzido abaixo.

```
Agencias    = (agencia_cod, agencia_nome, agencia_cidade, ativos)
Clientes    = (cliente_cod, cliente_nome, rua, cidade)
Depositos   = (agencia_cod, conta_numero, cliente_cod, saldo)
Emprestimos = (agencia_cod, cliente_cod, emprestimo_numero, quantia)
```

12. Selecione todos os clientes que possuem contas em agencia(s) que possui(m) o maior ativo.

```
Select cliente_nome
from clientes, depositos
where clientes.cliente_cod = depositos.cliente_cod and
      depositos.agencia_cod in
      (select agencia_cod
       from agencias
       where ativos >= all
            (select ativos
             from agencias)
```

13. Selecione o total de agencias por cidade, classificado por cidade.

```
Select cidade, count(agencia_cod) as total_agencias
from agencias
group by cidade
order by cidade
```

14. Selecione, por agências, o(s) cliente(s) com o maior saldo.

```
Select agencia_nome, cliente_nome, saldo
from clientes, depositos, agencias
where clientes.cliente_cod = depositos.cliente_cod and
      depositos.agencia_cod = agencia.agencia_cod and
      <agencia_cod, saldo> in
            (Select agencia_cod, max(saldo) as maior_saldo
             from depositos
             group by agencia_cod)
```

15. Selecione o valor médio de empréstimos efetuados por cada agência em ordem crescente das cidades onde estas agências se situam.

```
Select agencia_nome, avg(quantia) as valor_medio
from emprestimos, agencias
where emprestimos.agencia_cod = agencias.agencia_cod
group by agencia_nome
order by cidade
```

16. Selecione a(s) agência(s) que possui(m) a maior média de quantia emprestada.

```
Select agencia_nome
from agencias, emprestimos
where emprestimos.agencia_cod = agencias.agencia_cod
```

```
group by agencia_nome
having agv(quantia) >= all
      (select avg(quantia)
       from emprestimos
       group by agencia_cod)
```

17. **Selecione todas as agências situadas fora de Vitória que possuem a média de depósitos maior do que alguma agência localizada em Vitória.**

```
Select agencia_nome
from agencias, depositos
where depositos.agencia_cod = agencias.agencia_cod and
      agencia.cidade <> 'Vitória'
group by agencia_nome
having agv(saldo) > some
      (select avg(saldo)
       from depositos, empresas
       where depositos.codEmpresa = empresas.codEmpresa and
             empresas.cidade='Vitória'
       group by agencia_cod)
```

18. **Selecione o menor saldo de clientes, por agências.**

```
Select agencia_nome, min(saldo) as menor_saldo
from depositos, agencias
where depositos.agencia_cod = agencias.agencia_cod
group by agencia_cod
```

19. **Selecione o saldo de cada cliente, caso ele possua mais de uma conta no banco.**

```
Select cliente_nome, sum(saldo)
from clientes, depositos
where clientes.cliente_cod = depositos.cliente_cod
group by cliente_nome
having count(agencia_cod) >1
```

Considere o esquema abaixo para as questões que se seguem

```
Pessoas  = (CodPessoa, NomePessoa, Cidade, Chefe)
Empresas = (codEmpresa, NomeEmpresa, Cidade)
Trabalha = (CodPessoa, CodEmpresa, Salario)
```

20. **Excluir todas as pessoas que possuem salario = 1000.**

```
Delete pessoas where CodPessoa in
(select cod_pessoa from trabalha where salario = 1000)
```

21. **Excluir todas as pessoas que trabalham em empresas situadas em Vitória.**

```
Delete pessoas where CodPessoa in
(select cod_pessoa from trabalha, Empresas
 where trabalha.CodEmpresa=Empresas.CodEmpresa and cidade='Vitoria')
```

22. **Incluir na empresa de código '01' todos os moradores de Vitória com um salário=100.**

```
Insert into trabalha (codPessoa, codEmpresa, Salario)
Select codpessoa, '01', 100 from pessoas where cidade = 'Vitória'
```

23. **Uma determinada empresa de código 'x' vai contratar todos os funcionários da empresa de código 'y' que ganham acima de 1000, dando um aumento de salário de 10%. Faça comando(s) SQL para que tal transação seja efetuada.**

```
Insert into trabalha (codPessoa, codEmpresa, Salario)
Select codpessoa, 'x', salario*1.1
from trabalha
```

```
where codempresa='y' and salario >1000

Delete trabalha where codempresa = 'y' and salario >1000
```

24. Uma determinada empresa de código 'xyz' quer contratar todos que moram em Vitória e estão desempregados. Serão contratados com salário = 200. Faça comando(s) SQL para fazer tal transação.

```
Insert into trabalha (codPessoa, codEmpresa, Salario)
Select codp, 'xyx', 200
from pessoas
where cidade = 'Vitória' and codpessoa
      not in (select codpessoa from trabalha)
```

25. Fazer um comando SQL para ajustar o salário de todos os funcionários da empresa 'Campana' em 5%.

```
Update trabalha
Set salario = salario * 1.05
Where codEmpresa in
      (select codEmpresa
       from empresas
       where nomeEmpresa = 'Campana')
```

26. Todas as pessoas que moram em Colatina e trabalham na empresa 'Campana' deverão mudar para Vitória, devido aos requisitos do diretor da empresa. Faça comando(s) SQL para fazer esta atualização da cidade.

```
Update Pessoas
Set cidade = 'Vitória'
Where codPessoa in
      (select codPessoa
       from Pessoas, Trabalha, empresas
       where Pessoas.codPessoa = Trabalha.CodPessoa and
             Pessoas.cidade = 'Colatina' and
             Trabalha.CodEmpresa = Empresas.codEmpresa and
             nomeEmpresa = 'Campana')
```

3.13 O ESQUEMA BANCO

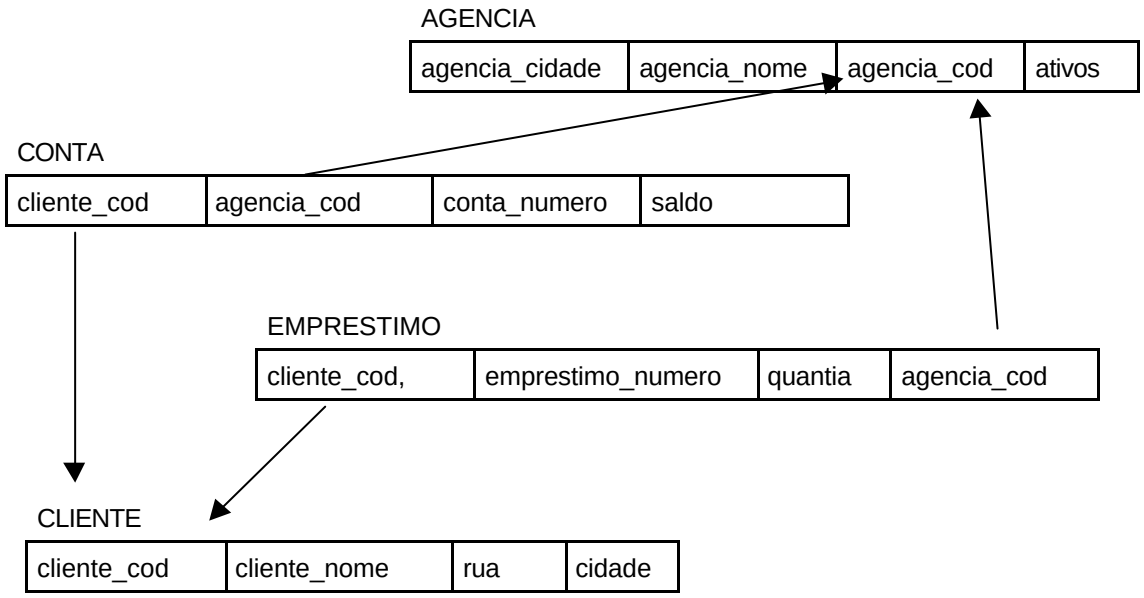
AGENCIA = (agencia_cod, agencia_nome, agencia_cidade, ativos)

CLIENTE = (cliente_cod, cliente_nome, rua, cidade)

CONTA = (agencia_cod, conta_numero, cliente_cod, saldo)

EMPRESTIMO = (agencia_cod, cliente_cod, emprestimo_numero, quantia)

De uma forma visual:



3.14 O ESQUEMA EMPRESA

Tabela EMPREGADO					
Nome	**RG**	**CIC**	**Depto**	**RG Supervisor**	**Salario**
João Luiz	10101010	11111111	1	NULO	3.000,00
Fernando	20202020	22222222	2	10101010	2.500,00
Ricardo	30303030	33333333	2	10101010	2.300,00
Jorge	40404040	44444444	2	20202020	4.200,00
Renato	50505050	55555555	3	20202020	1.300,00

Tabela DEPARTAMENTO		
Nome	**Numero**	**RG Gerente**
Contabilidade	1	10101010
Engenharia Civil	2	30303030
Engenharia Mecânica	3	20202020

Tabela PROJETO		
Nome	**Numero**	**Localizacao**
Financeiro 1	5	São Paulo
Motor 3	10	Rio Claro
Prédio Central	20	Campinas

Tabela DEPENDENTES				
RG Responsavel	**Nome Dependente**	**Nascimento**	**Relacao**	**Sexo**
10101010	Jorge	27/12/86	Filho	Masculino
10101010	Luiz	18/11/79	Filho	Masculino
20202020	Fernanda	14/02/69	Cônjuge	Feminino
20202020	Ângelo	10/02/95	Filho	Masculino
30303030	Adreia	01/05/90	Filho	Feminino

Tabela DEPARTAMENTO_PROJETO	
Numero Depto.	**Numero Projeto**
2	5
3	10
2	20

Tabela EMPREGADO_PROJETO		
RGEmpregado	**Numero Projeto**	**Horas**
20202020	5	10
20202020	10	25
30303030	5	35
40404040	20	50
50505050	20	35

Capítulo 4 PROCEDIMENTOS ARMAZENADOS

Procedimentos armazenados são rotinas definidas no banco de dados, visando um melhor desempenho de aplicações. Um procedimento armazenado, uma vez definido no banco, é executado no servidor toda vez que solicitado, reduzindo assim, o tráfego na rede. Não há uma padronização para tal, cabendo a cada banco uma especificação diferente.

Tipicamente os bancos de dados fornecem duas formas de armazenamento de procedimentos SQL: stored procedures e triggers. Procedures e triggers podem incluir controles de declarações SQL tais como controles de repetição (LOOP) e controles de desvio (IF, CASE).

Procedures e triggers são definidos no banco de dados, separados de qualquer aplicação do banco de dados. Esta separação traz várias vantagens como padronização, eficiência e seguraça.

- **Padronização:** Procedures e triggers permitem padronizar qualquer ação que seja necessária para um programa. A ação é codificada uma vez e armazenada no banco de dados. Assim o programa só precisa fazer um CALL para que o procedimento gere o resultado desejado. Outro aspecto importante é o da manutenção. Neste caso, as procedures permitem que as alterações no código SQL seja centralizadas em um só ponto..
- **Eficiência:** Quando usado em um BD implementado em uma rede, as procedures e triggeres são executadas na máquina servidora do banco de dados. Assim, os dados são acessados diretamente no banco de dados, sem que seja necessário grande uso de comunicação em rede. Isto significa que eles executam mais rápido e com menos impacto na performance da rede do que se eles fossem implementados em uma aplicação em uma das máquinas clientes. Alem disto, quando um trigger ou procedure é criado, é checada a sua sintaxe e, em alguns casos, são feitas otimizações nas consultas para que fiquem mais eficientes. Após isto o código SQL é compilado. Desta forma seu uso é muito mais rápido.
- **Segurança:** Procedures e triggeres permite que os mecanismos de segurança tenham sua operação otimizada.

4.1 STORED PROCEDURES

Procedimentos são chamados através de comandos como CALL ou EXEC. Usam passagem de parâmetros e retornam valores para o ambiente que os chamou. Dentro de um procedimento podemos chamar outros procedimentos.

4.1.1 Stored Procedure no SQL Server

A sintaxe geral para criação de procedures em SQL Server é:

```
Create procedure <nomestoreproc> (@param1 tipo, ...)
as
      <comandos SQL>
go
```

Exemplo

```
create procedure teste @Kall char(20)
```

```
as
      /*select @Kall = “eu”*/
      if  @Kall = ‘eu’
            print @Kall
      else
            print ‘teste’
go
```

Este procedimento é chamado da seguinte forma:

```
exec teste ‘eu’
```

4.1.2 Stored Procedure no SQL Anywhere

A sintaxe simplificada para criar uma procedure no SQL Anywhere é:

```
CREATE PROCEDURE procedure-name ([parâmetro , ...])
BEGIN
...
END
```

Abaixo mostramos uma tabela com uma série de comandos que são normalmente usados em procedures (e triggeres). Alguns destes comandos serão usados nos exemplos.

- Compound statements

```
BEGIN [ ATOMIC ]
   statement-list
END
```

- Conditional execution: IF

```
IF condition THEN
   statement-list
ELSEIF condition THEN
   statement-list
ELSE
   statement-list
END IF
```

- Conditional execution: CASE

```
CASE expression
   WHEN value THEN statement-list
   WHEN value THEN statement-list
   ELSE statement-list
END CASE
```

- Repetition: WHILE, LOOP

```
WHILE condition LOOP
   statement-list
END LOOP
```

- Repetition: FOR cursor loop

```
FOR
   statement-list
END FOR
```

- Break: LEAVE

```
LEAVE label
```

- CALL

```
CALL procname( arg, ... )
```

Exemplo

O procedimento abaixo permite que uma agência, **ag** , tenha seus ativos atualizados.

```
CREATE PROCEDURE Atualiza_Ativos (IN ag CHAR(20))
BEGIN
   update agencia
   set ativos =
            (select sum(saldo)
             from conta
             where conta.agencia_cod = agencia.agencia_cod)
   where (agencia_nome = ag);
END
```

Desta forma, uma chamada do tipo call Atualiza_Ativos ('Centro') atualizará apenas as agências cujo nome for Centro.

Exemplo

Uma forma mais sofisticada do exemplo anterior poderia ser a seguinte: caso o usuário do procedimento indique o nome da agência, então apenas os ativos dela devem ser atualizados mas caso ele não indique nome algum, então todas as agências devem ser atualizadas.

```
CREATE PROCEDURE Atualiza_Ativos (ag CHAR(20))
BEGIN
   if (ag = '') then
      update agencia
      set ativos =
         (select sum(saldo)
          from conta
          where conta.agencia_cod = agencia.agencia_cod);
   else
      update agencia
      set ativos =
         (select sum(saldo)
          from conta
          where conta.agencia_cod = agencia.agencia_cod)
      where (agencia_nome = ag);
   end if
END
```

Desta forma, uma chamada do tipo call Atualiza_Ativos ('') atualizaria todas as agências enquanto call Atualiza_Ativos ('Centro') atualizaria apenas as agências cujo nome fosse Centro.

Exemplo

Escreva um procedimento CriaClienteEspecial em que, dado os dados do cliente, crie automaticamente uma conta com saldo 1000,00 e um empréstimo equivalente de mesmo valor. O número da conta deve ser o maior número de conta disponível mais um.

```
CREATE PROCEDURE  CriaClienteEspecial (nome CHAR(20), ag INT)
BEGIN
   DECLARE cod_cli INT ;
   DECLARE cod_conta INT;

   -- Obtém o maior número do cliente e inclui
   set cod_cli = (select max(cliente_cod)from cliente);

   if (cod_cli = NULL) then –- verifica se é o primeiro cliente
      cod_cli = 0
   else
      cod_cli = cod_cli+1;

   insert into cliente
   values (cod_cli,nome,NULL,NULL);

   -- Obtém o maior número de conta da agência e inclui o novo cliente
   set cod_conta = (select max(conta_numero)
                    from conta
                    where conta.agencia_numero = ag);

   if (cod_conta = NULL) then –- verifica se é o primeiro
      cod_conta = 0
   else
      cod_conta = cod_conta +1;

   insert into conta
   values (cod_conta+1,1000,ag, cod_cli+1);

   insert into emprestimo
   values (ag, cod_cli+1, cod_conta+1001, 1000);

END
```

Exemplo

Escreva um procedimento Pagamento em que, dado o código de um empréstimo e o código de uma conta, o valor da conta seja usado para pagar o empréstimo.

```
ALTER PROCEDURE  Pagamento (con INT, emp INT)
BEGIN

   DECLARE valor_emp money;
   DECLARE valor_con money;

   -- Obtém o valor do empréstimo e da conta
   set valor_con = (select sum(saldo)
                    from conta
                    where conta_numero = con);

   set valor_emp = (select sum(quantia)
                    from emprestimo
                    where emprestimo_numero = emp);

   -- Atualiza o empréstimo e a conta
   if (valor_con > valor_emp) then
      Update CONTA
      set saldo = valor_con - valor_emp
      where conta_numero = con;

      Update EMPRESTIMO
      set quantia = 0
      where emprestimo_numero = emp;

   else
      Update EMPRESTIMO
      set quantia = valor_emp - valor_com
      where emprestimo_numero = emp;

      Update CONTA
      set saldo = 0
      where conta_numero = con;

   end if
end
```

4.2 GATILHOS (TRIGGERS)

Um gatilho é um procedimento executado automaticamente pelo sistema de banco de dados como efeito colateral de uma modificação no banco.

Triggers são associados com uma tabela específica do banco de dados. Eles são executados automaticamente (disparados) sempre que uma tupla da tabela associada for incluída, alterada ou excluída. Triggers não têm parâmetros e não podem ser executados através de um CALL como no caso de uma stored procedure. Entretanto, a ação de um trigger pode disparar outros triggeres. Também podemos chamar procedimentos de dentro de um trigger.

Para projetar um gatilho, precisamos:

- Especificar as condições sob as quais o gatilho deve ser acionado.
- Especificar as ações a serem tomadas quando um gatilho é executado.

Abaixo é mostrada uma sintaxe simplificada da criação de trigger no SQL Anywhere:

```
CREATE TRIGGER nome momento evento [, evento,..] ON tabela
[ REFERENCING [OLD AS old-name ] [ NEW AS new-name ] ]
[ FOR EACH { ROW | STATEMENT } ]

BEGIN
...
END

onde:
      momento: BEFORE|AFTER
      evento : DELETE|INSERT|UPDATE|UPDATE OF lista_de_colunas
```

De forma geral os bancos de dados permitem que os triggeres sejam disparados nas seguintes situações:

- DELETE: Quando uma tupla do banco é excluída.
- INSERT : Quando uma tupla do banco é incluída.
- UPDATE: Quando algum atributo de uma tupla do banco é alterado.
- UPDATE OF <lista de atributos> : Quando um ou mais atributos da <lista de atributos> é alterado. É uma forma de especifica que o trigger só será disparado se for alterada uma informação de uma coluna específica.

Pode-se especificar também se o trigger vai ser disparado antes (BEFORE) ou após (AFTER) a ocorrência de um UPDATE / DELETE / INSERT.

Por exemplo, suponha que seja especificado o disparo de um trigger sempre que houver uma atualização (UPDATE). Em algumas situações pode ser importante definir que o trigger será disparado antes da tupla ser efetivamente atualizada no banco. Em outras situações pode ser importante definir que o trigger será disparado após a tupla ser atualizada. Sendo assim, o SQL permite estas duas construções através do uso das cláusulas BEFORE e AFTER conforme veremos nos exemplos.

No SQL Anywhere podemos especificar se a ação do trigger será efetuada em relação à tupla que sofreu UPDATE / DELETE / INSERT ou se será em relação à tabela como um todo. No primeiro caso o trigger receberá como parâmetro os dados novos e antigos da tupla que está em foco (por exemplo a tupla que está sendo atualizada ou que está sendo incluída ou excluída). No segundo caso o trigger receberá como parâmetro a tabela nova (com os dados atualizados) e a tabela antiga (com os dados antes da atualização, inclusão ou exclusão). Para isto usamos as seguintes cláusulas:

- FOR EACH ROW : o trigger receberá como parâmetro a tupla.
- FOR EACH STATEMENT : o trigger receberá como parâmetro a tabela.

Exemplo

Suponha que, em vez de permitir saldos negativos, o banco trate o saque descoberto ajustando o saldo para zero e criando um empréstimo na quantia do saldo negativo. Para este empréstimo é dado um número igual ao da conta estourada. Neste caso, a condição para disparar o gatilho é uma atualização na relação CONTA que resulte em um valor de saldo negativo.

No SQL Anywhere este trigger ficaria da seguinte forma:

```
CREATE TRIGGER
saldo_negativo AFTER UPDATE OF saldo ON CONTA
REFERENCING OLD AS saldo_antigo
NEW AS saldo_novo
FOR EACH ROW
BEGIN
   IF saldo_novo.saldo < 0 then

      update CONTA AS C
      set C.saldo = 0
      where C.conta_numero = saldo_novo.conta_numero;

      INSERT INTO EMPRESTIMO values
      (saldo_novo.agencia_cod,
       saldo_novo.cliente_cod,
       saldo_novo.conta_numero+1000,
       - saldo_novo.saldo);

   END IF;

END
```

Exemplo

Escreva um gatilho que seja capaz de abrir uma conta sempre que o cliente que fizer um empréstimo em uma dada agência. O número da conta deverá ser o mesmo do empréstimo + 1000 e seu saldo, o mesmo do empréstimo.

```
create trigger deposita_emprestimo after insert on emprestimo
referencing new as novo_emprestimo
for each row
begin
  if
    (not exists
     (select*
      from conta
      where conta.agencia_cod=novo_emprestimo.agencia_cod
      and conta.cliente_cod=novo_emprestimo.cliente_cod))
  then
    insert into CONTA(agencia_cod,conta_numero,cliente_cod,saldo)
      values(
         novo_emprestimo.agencia_cod,
         novo_emprestimo.emprestimo_numero,
         novo_emprestimo.cliente_cod,
         novo_emprestimo.quantia)
  end if
end
```

4.3 OS CURSORES

Deve ficar claro que o uso dos cursores não se restrinjam à procedures e triggeres. De fato, o maior uso para este recurso é em programas feitos em linguagens de uso geral (como Pascal, C, C++, COBOL, etc) que fazem uso de bancos de dados.

Um cursor é usado para percorrer as linhas obtidas através de uma consulta a um banco de dados (um SELECT... FROM...WHERE). Os cursores são manipulados de forma semelhante à manipulação de arquivos em uma linhagem de programação.

Os seguintes passos devem ser seguidos para manipular um cursor:

1 O cursor é declarado para um select específico usando o comando DECLARE.

2 O cursor é aberto usando o comando OPEN.

3 O comando FETCH é usado para obter o resultado de uma linha e atribuí-lo ao cursor.

4 Usualmente os registros (linhas) são percorridos (FETCH) até que um erro Row Not Found é retornado.

5 O cursor é então fechado usando o comando CLOSE.

Exemplo

Esta procedure retorna o nome da agência que possui maior saldo. Esta procedure não leva em conta a possibilidade de haver mais de uma agência com o mesmo saldo(maior). A idéia é apenas exemplificar o uso de cursores.

```
CREATE PROCEDURE MaiorAtivos (OUT NomeAgencia CHAR(20))
BEGIN
   -- 1. Declare the "error not found" exception
   DECLARE err_notfound EXCEPTION FOR SQLSTATE '02000';

   -- 2. Variáveis auxiliares
   DECLARE MaiorAtivo MONEY;
   DECLARE AtivoCorrente MONEY;
   DECLARE AgenciaCorrente CHAR(20);

   -- 3. Declara o cursor
   DECLARE AgenciaCorrente CURSOR FOR
      SELECT agencia_nome, ativos
      FROM agencia;

   -- 4. Inicia valores
   SET MaiorAtivo = 0;

   -- 5. Abre o cursor
   OPEN AgenciaCorrente;

   -- 6. Percorre as linhas da seleção feita
   AgenciaLoop:
   LOOP
      FETCH NEXT AgenciaCorrente INTO AgenciaCorrente, AtivoCorrente;

      IF SQLSTATE = err_notfound THEN
         LEAVE AgenciaLoop;
      END IF;

      IF AtivoCorrente > MaiorAtivo THEN
         SET MaiorAtivo = AtivoCorrente;
         SET NomeAgencia = AgenciaCorrente;
      END IF;
```

```
    END LOOP AgenciaLoop;

    -- 7. Fecha o cursor
    CLOSE AgenciaCorrente;

END
```

4.4 SQL EMBUTIDA

A forma de linguagem de programação de banco de dados mais largamente utilizada é a SQL embutida em uma linguagem de programação de propósito geral, referenciando uma linguagem hospedeira. Na SQL embutida, todos os processamentos de consulta são executados pelo sistema de banco de dados. O resultado desta consulta é então tornado disponível ao programa, uma tupla por vez.

O SQL Embutido trabalha inserido no código de uma linguagem de programação tradicional como COBOL, PL/I, C, etc... que recebe o nome de linguagem hospedeira. A idéia principal é unir todo o poder expressivo das linguagens de programação de propósito geral, à versatilidade do modelo relacional dos bancos de dados modernos.

A sintaxe da programação é essencialmente a mesma para todas as linguagens hospedeiras, diferenciando principalmente nas definições das variáveis de trabalho internas de cada uma.

Um programa SQL embutido precisa ser processado por um pré-processador especial antes da compilação. Esse pré-processador é geralmente fornecido pelo banco de dados. As requisições SQL embutidas são substituídas por declarações e chamadas de procedimentos da linguagem hospedeira, os quais permitem a execução em conjunto com acessos do banco de dados. Então, o programa resultante é compilado pelo compilador da linguagem hospedeira.

O SQL Embutido fornece uma maneira mais interativa de trabalho para o programador, propiciando uma fase de testes e depurações muito mais prática. Além disso, padroniza o acesso à base de dados, pois em qualquer linguagem hospedeira o SQL Embutido é o mesmo, salvo pequenas modificações. E, como se sabe, SQL é um padrão estabelecido.

Por outro lado, a união entre a linguagem hospedeira e o SQL Embutido se dá de uma forma pouco definida, tendo cada uma várias restrições de acesso aos objetos da outra. Isto causa uma desestruturação da programação.

Podemos considerar alguns pontos importantes na utilização da linguagem:

- Toda instrução SQL embutida é identificada pela cláusula **`EXEC SQL`** de modo a diferenciá-la das instruções da linguagem hospedeira e ser encontrada pelo pré-processador do banco de dados;
- A transferência de dados entre a linguagem principal e o SQL Embutido é feita por variáveis internas, precedidas de dois pontos (:) para identificá-las em meio aos objetos do SQL;
- Todas as tabelas do banco de dados utilizadas no programa devem ser declaradas na cláusula EXEC SQL DECLARE;
- Toda instrução SQL retorna uma informação numa área de comunicação (**SQLCA**), declarada anteriormente com a cláusula EXEC SQL INCLUDE SQLCA. Esta área é principalmente utilizada para manipulação de erros e detecção de fim de registros.

Vamos tratar separadamente cada um destes pontos.

Afim de identificar requisições SQL embutidas para o pré-processador, é usado o comando EXEC SQL, que tem a forma:

```
exec sql < comandos sql> end exec
```

O comando **SQL include** é colocado no programa para identificar o lugar onde o pré processador deve inserir as variáveis especiais usadas para comunicação entre o programa e o sistema de banco de

dados. As variáveis da linguagem hospedeira podem ser usadas dentro de comandos SQL embutidos, mas eles precisam ser precedidos de (:) para distinguí-las das variáveis SQL.

Para escrever uma consulta relacional é usado o comando **declare cursor**. O resultado da consulta não é ainda computado. Em vez disso, o programa precisa usar os comandos **open** e **fetch** para obter as tuplas resultado.

4.4.1 Principais comandos da SQL Embutida

A SQL Embutido fornece comandos para:

- **Inclusão de áreas de comunicação com a linguagem hospedeira:**
  `EXEC SQL INCLUDE SQLCA`

- **Declaração da estrutura**

  É a declaração das variáveis do programa que receberão/enviarão valores às instruções SQL:
  `EXEC SQL BEGIN DECLARE SECTION`

- **Conexão e desconexão com o Banco de Dados acessado por ela:**

  Conexão:
  ```
  EXEC SQL CONNECT 'mydb'
  ```
  Desconexão:
  ```
  EXEC SQL DISCONNECT
  ```

- **Tratamento de cursores:**

  Declaração:
  ```
  EXEC SQL DECLARE 'mycur' CURSOR FOR 'SELECT colum1, colu...
  ```
  Abertura:
  ```
  EXEC SQL OPEN 'mycur' / EXEC SQL CLOSE 'mycur'
  ```
  Leitura:
  ```
  EXEC SQL FETCH 'mycur' INTO : var1, :var2( DECLARE SECTION * )
  ```

- **Tratamento de exceções:**
  `EXEC SQL WHENEVER 'condição' 'ação'`

  Onde condição pode ser: `SQLERROR`, `NOT FOUND`, `SQLMESSAGE`

  E ação pode ser: `CONTINUE`, `STOP`, `GOTO 'label'`, `CALL 'procedure'`

- **Gravar alterações exercidas no banco:**
  `EXEC SQL COMMIT`

Exemplo

Assuma que tem uma variável quantia da linguagem hospedeira e que desejamos encontrar o nome e a cidade de clientes com mais de quantia em alguma conta. Isto pode ser escrito assim:

```
exec sql
   declare c cursor for
      select cliente-nome, cliente-cidade
```

```
      from clientes, depositos
      where clientes.cliente-cod=depositos.cliente-cod
            and saldo>:quantia
end-exec
```

A variável c nesta expressão é chamada de cursor para a consulta. Esta variável é usada para identificar a consulta no comando open, que causa a avaliação da consulta, e o comando fetch, que coloca os valores de uma tupla em variáveis da linguagem hospedeira.

Uma vez declarado, devemos abrir o cursor.

```
      Exec sql Open c end-exec
```

Isto leva o sistema de banco de dados a executar a consulta e a salvar os resultados em uma relação temporária.

Uma série de comandos fetch é executada para tornar as tuplas do resultado disponíveis para o programa. O comando fetch requer uma variável da linguagem hospedeira para cada atributo da relação resultante.

Exemplo

```
      exec sql fetch c into :cn, :cc end-exec
```

O programa pode então manipular as variáveis cn e cc usando os recursos da linguagem hospedeira.

Uma única requisição fetch retorna apenas uma tupla. Para obter todas as tuplas do resultado, o programa precisa conter um loop para iteragir sobre todas as tupla. Uma variável SQLCODE é ajustada para indicar que não há mais nenhuma tupla a ser processada.

O comando close precisa ser usado para dizer ao sistema de banco de dados que feche o cursor especificado.

```
      Exec sql close c end-exec
```

Uma vez fechado, deve ser liberada a área de memória alocada para o cursor.

```
      Exec sql deallocate c end-exec
```

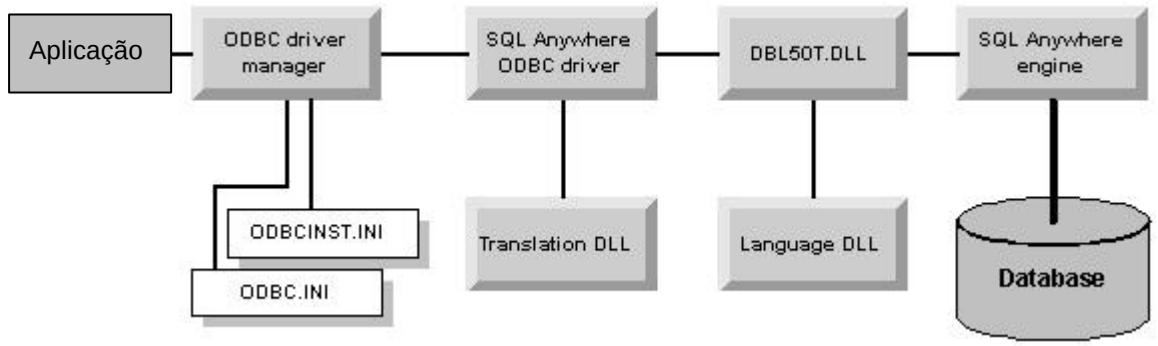
As expressões SQL embutidas para modificações do banco de dados (update, insert e delete) não retornam um resultado. Assim, elas são mais fáceis de expressar. Uma requisição de modificação do banco de dados toma a forma:

```
      Exec sql <qualquer update, delete ou insert> end-exec
```

4.4.2 Acesso via ODBC

Uma das formas mais usadas de SQL embutida em ambiente Microsoft é usando ODBC (Open Database Connectivity).

Abaixo apresentamos um esquema da forma como um programa se comunica com um banco de dados (no caso o SQL Anywhere). (a figura foi extraída da documentação on-line do SQL Anywhere).



Abaixo apresentamos um exemplo simples de código C que acessa um banco de dados via ODBC. O programa executa o banco de dados chamado Banco que foi previamente construído usando SQL Anywhere (poderia ser feito em qualquer banco inclusive Access).

O programa imprime na tela o resultado de um `select * from cliente`.

```
//
// Teste para acesso à banco ODBC
// Chico Rapchan - Banco de Dados 1999.
//

#include <stdio.h>

#define ODBC_WINNT  // Define o tipo do sistema operacional
#include "odbc.h"

void main (int argc, char * argv[])
{
   int   i;               // Usado como contador nos for's
   int   iNumCol;         // Número de colunas
   SWORD swTipoCampo;     // Tipo do campo
   UCHAR szNomeCampo[30]; // Nome do campo
   UCHAR szMsgErro[100];  // Armazena a mensagem de erro retornada
   RETCODE CodErro;       // Valor de retorno das funções ODBC

   // Variáveis especificadas pela documentação ODBC
      HENV    env;  // Handle para o ambiente (área de memória)
      HDBC    dbc;  // Handle referente ao banco ODBC
      HSTMT   stmt; // Handle referente à operação SQL executada

   // Faz alocações de memória
      SQLAllocEnv(&env);
      SQLAllocConnect(env, &dbc);
```

```
// Estabelece uma conexão com o banco onde
// Banco é o nome DSN (Data Source Name) dado no ODBC
// dba é o userid e dba é a senha

SQLConnect( dbc, (UCHAR *)"Banco", SQL_NTS,
                 (UCHAR *)"dba",   SQL_NTS,
                 (UCHAR *)"dba",   SQL_NTS);

SQLSetConnectOption( dbc, SQL_AUTOCOMMIT, FALSE );

// Faz alocações de memória
if( SQLAllocStmt( dbc, &stmt ) == SQL_ERROR )
{
   SQLError( env, dbc, SQL_NULL_HSTMT, NULL, NULL,(UCHAR *)&szMsgErro,
             sizeof(szMsgErro), NULL );
   printf("Erro no SQLAllocStmt: %s\n", szMsgErro );
   return;
}

// Executa o comando SQL
SQLExecDirect(stmt,(UCHAR *)"select * from cliente",SQL_NTS);

// Obtém o número de colunas
SQLNumResultCols(stmt,(short *)&iNumCol);


// Coloca o cabeçalho na tela
printf ("\n");
for (i=1; i<= (int)iNumCol; i++)
{
   CodErro = SQLDescribeCol(stmt,i,(UCHAR *)&szNomeCampo,
                            sizeof(szNomeCampo),NULL,
                            &swTipoCampo,NULL,NULL,NULL);
   printf ("<%s> ",szNomeCampo);
}

// Executa este loop até que fazer o último Fetch
while (TRUE)
{
   // Faz um Fetch
   CodErro = SQLFetch(stmt);

   // Se houve sucesso no Fetch
   if (CodErro == SQL_SUCCESS || CodErro == SQL_SUCCESS_WITH_INFO)
   {
      char  szValor[30]; // Conteúdo do campo se for string
      int   iValor;      // Conteúdo do campo se for int

      // Percorre todas as colunas imprimindo os valores
      for (i=1; i<= (int)iNumCol; i++)
      {
         // Obté dados da coluna
         CodErro = SQLDescribeCol(stmt,i,
                                  (UCHAR *)&szNomeCampo,
                                  sizeof(szNomeCampo),
                                  NULL,&swTipoCampo,
                                  NULL,NULL,NULL);

         // Faz trata caso o campo seja numerico ou string.
         if (swTipoCampo == SQL_INTEGER)
         {
```

```
                SQLGetData(stmt,i,SQL_C_DEFAULT,&iValor,sizeof(iValor),NULL);
                printf("%-5i", iValor);
             }
             else
             {
                SQLGetData(stmt,i,SQL_C_DEFAULT,&szValor,sizeof(szValor),NULL);
                printf("%-20s", szValor);
             }
          }
          printf("\n");
       }
       else  // Se chegou ao fim, sai do while
       break;

   } //while do Fetch

   /* Confirma a operação */
   SQLTransact( env, dbc, SQL_COMMIT );

   // Libera a memória e desconecta do banco
   SQLFreeStmt( stmt, SQL_DROP );
   SQLDisconnect( dbc );
   SQLFreeConnect( dbc );
   SQLFreeEnv( env );
}
```

Capítulo 5 PROJETO DE BANCOS DE DADOS RELACIONAIS

O objetivo do projeto de um banco de dados relacional é gerar um conjunto de esquemas de relações que nos permita armazenar informações em redundância desnecessária e, ainda, nos permita recuperar informações facilmente.

A forma de se fazer isso é projetar esquemas de bancos de dados na forma normal apropriada.

5.1 DEPENDÊNCIA FUNCIONAL

A dependência funcional é uma generalização da noção de chave. São restrições ao conjunto de relações válidas. Elas permitem expressar determinados atos em um banco de dados relativos ao problema, à empresa ou ao negócio que desejamos modelar.

De fato, a noção de dependência funcional generaliza a noção de superchave. Usando a notação de dependência funcional dizemos que K é uma superchave de R se k → R.

Considere o esquema:

Esquema_info_emprestimo = (nome_agencia, numero_emprestimo, nome_cliente_total)

5.2 – FORMAS NORMAIS

O processo de **normalização** pode ser visto como o processo no qual são eliminados esquemas de relações (tabelas) não satisfatórios, decompondo-os, através da separação de seus atributos em esquemas de relações menos complexas mas que satisfaçam as propriedades desejadas.

O processo de normalização como foi proposto inicialmente por Codd conduz um esquema de relação através de um bateria de testes para certificar se o mesmo está na **1ª**, **2ª** e **3ª Formas Normais**. Estas três Formas Normais são baseadas em dependências funcionais dos atributos do esquema de relação.

5.2.1 1ª Forma Normal

Primeira Forma Normal: Uma relação se encontra na primeira forma normal se todos os domínios de atributos possuem apenas valores atômicos (simples e indivisíveis), e que os valores de cada atributo na tupla seja um valor simples. Assim sendo todos os atributos compostos devem ser divididos em atributos atômicos.

A **1ª Forma Normal** prega que todos os atributos de uma tabela devem ser **atômicos** (indivisíveis), ou seja, não são permitidos atributos multivalorados, atributos compostos ou atributos multivalorados compostos. Leve em consideração o esquema a seguir:

CLIENTE

1. Código
2. { Telefone }
3. Endereço: (Rua, Número, Cidade)

gerando a tabela resultante:

		Telefone 1	Endereço		
Cliente	Código	Telefone *n*	Rua	No	Cidade

sendo que a mesma não está na 1ª Forma Normal pois seus atributos não são atômicos. Para que a tabela acima fique na 1ª Forma Normal temos que eliminar os atributos não atômicos, gerando as seguintes tabelas como resultado:

Cliente	Código	Rua	Número	Cidade

Cliente_Telefone	Código_Cliente	Telefone_Cliente

5.2.2 2ª Forma Normal

Segunda Forma Normal: Uma relação se encontra na segunda forma normal quando estiver na primeira forma normal e todos os atributos que não participam da chave primária são dependentes desta. Assim devemos verificar se todos os atributos são dependentes da chave primária e retirar-se da relação todos os atributos de um grupo não dependente que dará origem a uma nova relação, que conterá esse atributo como não chave. Desta maneira, na segunda forma normal evita inconsistências devido à duplicidade.

A **2ª Forma Normal** prega o conceito da **dependência funcional total**. Uma dependência funcional **X → Y** é **total** se removemos um atributo **A** qualquer do componente **X** e desta forma, a dependência funcional deixa de existir. A dependência funcional **X → Y** é uma **dependência funcional parcial** se existir um atributo **A** qualquer do componente **X** que pode ser removido e a dependência funcional **X → Y** não deixa de existir.

Veja a seguinte dependência funcional

{ RG_Empregado, Número_Projeto } → Horas

Esta dependência implica que o **RG do empregado** mais o **número do projeto** define de forma única o número de horas que o empregado trabalhou no projeto.

Esta é uma dependência funcional total, pois se removermos o atributo **RG_Empregado** ou o atributo **Número_Projeto**, a dependência funcional deixa de existir.

5.2.3 3ª Forma Normal

Terceira Forma Normal: Uma relação estará na terceira forma normal, quando estiver na primeira forma norma e todos os atributos que não participam da chave primária são dependentes desta porém não transitivos. Assim devemos verificar se existe um atributo que não depende diretamente da chave, retirá-lo criando uma nova relação que conterá esse grupo de atributos, e defina com a chave, os atributos dos quais esse grupo depende diretamente.

A **3ª Forma Normal** prega o conceito de **dependência transitiva**. Uma dependência funcional **X→Y** em uma tabela **T** é uma dependência transitiva se existir um conjunto de atributos **Z** que não é um subconjunto de chaves de **T** e as dependências **X→Z**, **Z→Y**, são válidas. Considere a seguinte tabela como exemplo:

Empregado	RG	Nome	Nº_Departamento	Nome_Depto	RG_Ger_Depto

onde temos a seguinte dependência transitiva:

RG → { Nome_Depto, RG_Ger_Depto }

RG → Nº_Departamento

Nº_Departamento → { Nome_Depto, RG_Ger_Depto }

Porém, verifique o caso da tabela abaixo:

Empregado	RG	CIC	Nome	Nº_Funcional

RG → { Nome, Nº_Funcional }

RG → CIC

CIC → { Nome, Nº_Funcional }

não é valida pois o atributo CIC é uma chave candidata.

Uma tabela está na 3ª Forma Normal se estiver na 2ª Forma Normal e não houver dependência transitiva entre atributos não chave.

5.2.4 Resumo

- **1ª forma normal:** Nenhum atributo de uma tabela pode ser multivalorado.
- **2ª forma normal:** Nenhum atributo pode ser dependente de parte da chave primária de uma tabela. ex: (nome, logradouro, dt_nascimento, cep). Solução: colocar (nome, logradouro, dt_nascimento) em uma tabela e (logradouro, cep) em outra tabela.
- **3ª forma normal** : Nenhum atributo pode ser dependente de outro atributo não chave primária de uma tabela.

Capítulo 6 BIBLIOGRAFIA

SILBERSCHATZ, Abraham; KORTH, Henry; SUDARSHAN, S. *Sistema de Banco de Dados*. 3ª Edição. Makroh Books. 1999.

DATE, C. J. *Introdução a sistemas de banco de dados***.** Rio de Janeiro: Campus, 1990.

BOCHENSKI, Barbara. *Implementando sistemas cliente/servidor de qualidade***.** São Paulo: McGraw-Hill, 1995.

VASKEVICTH, DAVID. *Estratégia cliente/servidor***.** São Paulo: Berkeley, 1995.