**Banco de Dados:** Coleção ou conjunto de dados relacionados. No nosso contexto, um banco de dados tem as seguintes propriedades:

1. Representa algum aspecto ou contexto do mundo (universo de discurso);
2. É uma coleção logicamente coerente, com significado inerente;
3. É projetado, construído e populado com dados para uma finalidade específica.

O uso de banco foi motivado pela necessidade de armazenar informações de modo permanente (programa para gerenciar contatos telefônicos foi a primeira sugestão para armazenar as informações de modo permanente, provavelmente seriam armazenadas em arquivos). Inicialmente, as aplicações usavam arquivos. Para o exemplo da agenda de contatos, poderíamos ter em um arquivo binário:

|  |  |  |
| --- | --- | --- |
| Nome (100 bytes) | Data.Nasc (20 bytes) | Telefone (20 bytes) |

Este arquivo deveria ter, pelo menos, 140 bytes.

Em nosso arquivo binário, saberíamos que após o início dos registros, temos um registro a cada 140 bytes. Nosso programa usaria essa informação para acessar os registros. Nosso programa também acessaria o telefone contando 120 bytes do início do registro. Ao armazenar dados em arquivos, é necessário escrever um código específico para gerenciá-los.

Vamos supor que desejemos acrescentar a informação de e-mail em cada registro. E agora?

1. Seria preciso criar um novo campo para cada registro no arquivo e modificar (atualizar) todos os programas que fazem leitura e gravação no mesmo.
2. A segunda opção seria criar um novo arquivo só com nomes e e-mails (informações repetidas, pode causar inconsistência).

|  |  |
| --- | --- |
| Nome | Email |

E se for preciso obter uma lista com os maiores de 18?

Seria preciso criar um novo programa para ler a base de dados e fazer filtragem.

Múltiplos acessos ao arquivo de dados podem gerar problema se forem feitos simultaneamente.

**Um Sistema Gerenciador de Banco de Dados** (ou SGBD) é uma coleção de programas que permitem a criação, utilização e a manutenção de um Banco de Dados. Os SGBD surgem como alternativa à manutenção de dados por meio de arquivos, fazendo consultas nestes bancos de dados.

1. Gravar e recuperar dados;
2. Evitar acesso indevido/simultâneo;
3. Metadados (data, nome) – Descreve os dados.

Os SGBD surgem como alternativa à manutenção de dados por meio de arquivos. As vantagens de usar SGBD’s incluem:

* SGBD possui um catálogo com metadados que descrevem a estrutura do Banco de Dados (Podem ser utilizados para entender a estrutura do banco de dados).
* Permite o isolamento entre dados e programas.

Na maioria dos casos, mudanças na estrutura dos dados não requer modificações nos programas que fazem acesso aos mesmos. Com arquivos ordinários, seria necessário modificar os programas.

* Suporte para múltiplas visões de dados: Nem todos os usuários precisam ter acesso a todo o banco de dados, mas apenas a um subconjunto dele. Alguns usuários só precisam ler informação, enquanto outros podem ler e escrever.
* Compartilhamento de dados e operações entre múltiplos usuários. SGBD’s incluem controle de concorrência para garantir o correto acesso e atualização dos dados.
* Controle de redundância dos dados.
* Acesso eficiente aos dados.
* Ferramentas de backup e recuperação.
* Facilidade para lidar com relacionamentos complexos entre dados (1 professor dá várias matérias e 1 matéria tem 1 professor).
* Integridade dos dados (Ano de nascimento não pode ser negativo, dessa forma, o programa não permite ao usuário armazenar a data negativa).

Alguns exemplos de SGBD: MySQL, SQL Server, Oracle, PostgreSQL, Firebird, Access...

A modelagem é o primeiro passo na criação de um banco de dados. Compreender bem os processos modelados é fundamental para a criação de um modelo adequado e eficiente, que servirá de base para etapas posteriores de engenharia de software, na construção de uma aplicação.

**Modelo de dados**: Coleção de conceitos que podem ser usados para a estrutura de um banco de dados. Modelos de dados conceituais ou de alto nível se valem dos seguintes conceitos:

* **Entidade**: Representa um objeto ou conceito do mundo real (um veículo, um funcionário, uma turma, um projeto, etc.) no Banco de Dados. Entidades são descritas por meio de atributos.
* **Atributos**: Propriedade de interesse que ajuda a descrever uma entidade. Exemplo: nome de um funcionário, quilometragem de um veículo, um professor de uma turma. Note que o representante de uma entidade pode ser usado como atributo de outra.
* **Relacionamento**: Associação entre duas ou mais entidades. Exemplo: Um professor pode estar associado a *n* turmas, mas cada turma só pode estar associada a um professor.

**Modelo Entidade Relacionamento**: O MER é um modelo de dados conceitual de alto nível bastante popular na área de BD. Utiliza DER. Exemplo: Vamos armazenar informações sobre departamentos de uma empresa. O consumo dos diferentes departamentos é então modelado com a entidade departamento.

|  |
| --- |
| Departamento |
| * ID
 |
| * Nome
 |
| * Missão
 |
| * Localização
 |
| * Func.Respons
 |

Aqui, introduzimos um identificador auxiliar para o departamento (ID). A ideia é que esse ID seja único para cada departamento, de modo a possibilitar a sua identificação.

Atributos (nome, missão, localização e func.respons). No banco de dados, essa entidade será representada por meio de uma tabela, onde cada atributo estará em uma coluna. E em cada linha, teremos instâncias de departamento.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | Nome | Missão | Localização | Func. Respons |
| 001 | RH | Pessoas | 304 – A | 0107 |
| 002 | Financeiro | Finanças | 502 – B | 0025 |
| 003 | Produção | Produzir | I – 444 | 0235 |
| 004 | Inovação | Inovar | NULL | NULL |

 Podemos também modelar a entidade funcionários:

[IMAGEM 1]

Um departamento pode conter *n* funcionários como membros. Mas cada funcionário só pode estar em um departamento. Cada funcionário só pode ser responsável por um departamento.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Matrícula | CPF | Dt. Nasc | Endereço | Depto | Cargo |
| 0025 | 088... | 01/01/91 | Rua... | 002 | ... |

Note que, nos referimos ao departamento através do seu identificador único (ID). Desse modo, não precisamos repetir as informações do departamento nessa tabela e evitamos redundância.

Observe ainda que a matrícula do funcionário na empresa também funciona como identificador único, e que este campo foi usado na entidade (tabela) departamento para referenciar o funcionário responsável. Esses identificadores únicos são denominados chaves. Uma chave é um subconjunto mínimo de atributos (cujos valores permitem identificar unicamente um representante de uma entidade).

Uma chave pode ser composta por mais de um atributo em conjunto (**chaves compostas**) ou por um único atributo (**chaves simples**). Na tabela funcionários, ambos os campos matrícula e CPF poderiam funcionar como chave simples. Por isso, denominamos estes campos como **chaves candidatas**. No nosso exemplo, escolhemos matrícula para ser a chave. No modelo ER, cada atributo chave aparece sublinhado.

As chaves que identificam instâncias da própria entidade são denominadas como **chaves primárias** (campos ID do departamento e Matrícula em funcionário). Por sua vez, chaves que identificam instâncias de outra entidade são denominadas **chaves estrangeiras** (func.responsável de departamento e departamento de funcionário).

Quando uma entidade não possui atributos chave, a mesma é denominada **Entidade Fraca**.

[IMAGEM 2]

Nesse caso, dependente é uma entidade fraca. Toda entidade fraca precisa estar ligada a uma entidade forte (funcionário). É como se a entidade fraca “pegasse carona” na chave primária da forte, pois para se diferenciar os dependentes, usa-se a chave primária de funcionário junto com o nome de dependente.

Chave primária de funcionário: Matrícula.

Chave primária de dependente: Matrícula + nome.

* **Atributo simples**: Atributos cujos valores não podem ser subdivididos (CPF, RG, sexo, etc.).
* **Atributo composto**: Atributos cujos valores podem ser subdivididos em atributos simples (Ex: endereço pode ser dividido em rua, número, cidade, bairro, etc. Nome pode ser dividido entre primeiro nome e sobrenomes).
* **Atributo monovalorado**: Possui um único valor para cada representante da entidade. Ex: CPF, RG, data.nasc.
* **Atributo multivalorado**: Pode apresentar um conjunto de valores para cada representante da entidade. Ex: Telefone. A pessoa pode ter vários telefones, endereços.
* **Atributo armazenado**: Atributo da entidade realmente armazenado no Banco de Dados. Ex: Nome, data de nascimento, etc.
* **Atributo derivado**: Atributo cujo valor pode ser deduzido a partir dos valores de outros atributos. Ex: A idade pode ser inferida a partir da data.

**Grau de relacionamento**: Número de entidades envolvidas no mesmo. Dados um professor e um aluno monitor, existe no máximo uma turma que esse aluno monitora. Uma turma pode ter diversos professores e alunos monitores.

[IMAGEM 3].

Exercício de modelagem de banco de dados:

[imagem 4]

**Banco de dados relacionais**:

Se tornaram padrão a partir dos anos 1970 devido a sua simplicidade e embasamento matemático. É baseado no **modelo de dados relacional**.

**Modelo relacional**: Representa o Bando de Dados como uma coleção de relações. Podemos pensar em uma relação como algo semelhante a uma tabela de valores, onde cada linha representa uma coleção de valores relacionados. Desse modo, a relação pode ser vista como uma entidade, as linhas da tabela como instâncias (exemplos) da entidade e as colunas como atributos das entidades.

Formalmente, cada linha da tabela é chamada de tupla ou n-tupla (lista composta por *n* valores). Cada componente da tupla pode ser representado por um dado de um departamento tipo sujeito a um domínio de possíveis valores.

|  |
| --- |
| Pessoa |
| Nome: String |
| CPF: String |
| Sexo: Char |
| Dt.Nasc: Date |

|  |  |  |  |
| --- | --- | --- | --- |
| Nome | CPF | Dt. Nasc | Sexo |
| Jéssica G | 088... | 01/01/91 | F |
| Camila C | 078... | 12/05/92 | F |

Tupla: Vetor, lista com n posições.

Um atributo do domínio pode receber um nome, um tipo de dado e um formato. Ex: O atributo nome pode estar sobre o domínio Nomes Próprios, como tipo String. O atributo CPF pode estar em um domínio chamado CPF – Brasil, como tipo sequência, de *n* dígitos.

Um esquema relacional (R, usado para descrever a relação R), é indicado por:

R (A1, A2, ..., An).

Onde R = Nome da relação. A1 ... An = Atributos. E a relação é de grau n.

Dom (Ai) representa o domínio, isto é, o conjunto de valores possíveis do atributo Ai. O grau ou granularidade da relação é definido como o seu número de atributos. Para o nosso exemplo:

Pessoa (Nome, CPF, Sexo, Dt. Nasc) pode-se definir a relação de forma mais detalhada indicando os tipos dos atributos.

Pessoa (Nome: String, CPF: String; Sexo: Char, Dt. Nasc: Date). A definição de uma relação não inclui uma ordem específica para as tuplas. Assim, as tuplas da tabela Aluno poderiam vir em qualquer ordem que o **estado corrente** da relação seria o mesmo. No modelo relacional, cada valor na tupla representa um atributo atômico. Assim, atributos multivalorados **não** são permitidos nesse modelo.

No modelo relacional, as relações podem representar tanto fatos sobre entidades quanto fatos sobre relacionamentos.

**Restrições em modelo de dados relacionais:** Na prática, costuma haver muitas restrições sobre os valores de atributos das instâncias das entidades modeladas em um BD. Essas restrições vêm das regras de negócio do mundo real, e podem ser divididas em 3 categorias principais.

1. **Restrições inerentes baseadas no modelo, ou implícitas**: Restrições que são inerentes ao modelo de dados (no nosso exercício da última aula, cada projeto só pode ter um responsável);
2. **Restrições baseadas em esquemas ou explícitas**: Restrições que podem ser expressas diretamente nos esquemas do modelo de dados. Comumente, são especificadas na linguagem DDL (Data Definition Language).
3. **Restrições baseadas na aplicação, restrições semânticas ou regras de negócio:** Restrições que não podem ser expressas diretamente nos esquemas do modelo de dados, e, assim, devem ser impostas pelos programas de aplicação.

 Bancos de dados relacionais: Definimos uma superchave de uma relação R como um conjunto de atributos tal que não podem haver duas ou mais tuplas (instâncias da relação) com a mesma combinação de valores para os mesmos (exclusividade de valores de atributos). Desse modo, superchaves podem ser usadas como identificadores de instância de uma relação (entidade). Uma superchave mínima ou chave, é uma superchave da qual não se pode remover qualquer um dos atributos que a compõem sem violar a restrição de exclusividade de valores de atributos. Superchaves mínimas são denominadas chaves candidatas. Dentre as chaves candidatas de uma relação, uma é escolhida arbitrariamente para ser a chave primária. Se não houverem chaves candidatas, sempre é possível introduzir um atributo artificial para funcionar como chave:

Ex: Cadastro de uma agenda:

|  |  |
| --- | --- |
| Nome | Telefone |
| S. Silva | 3236-4962 |
| M. Silva | 3234-6631 |
| M.Silva | 3234-6631 |
|  |  |

Supondo que podem haver duas pessoas diferentes com o mesmo nome e telefone, não há superchave candidata. Podemos então introduzir um atributo artificial que servirá como chave primátia.

|  |  |  |
| --- | --- | --- |
| ID (atributo chave inserido artificialmente) | Nome | Telefone |
| 1 | S. Silva | 3236-4962 |
| 2 | M. Silva | 3234-6631 |
| 3 | M.Silva | 3234-6631 |

Em alguns casos utiliza-se a técnica do atributo chave artificial quando a superchave é composta de 1 ou + atributos. Assim, os relacionamentos envolvendo chaves estrangeiras só usarão um atributo. Por exemplo: Cadastro de Municípios.

|  |  |
| --- | --- |
| Municípios | Estados |
| Bom Jesus | RN |
| Bom Jesus | RS |
| Bom Jesus | SC |
|  |  |

Para se evitar a chave composta, pode-se utilizar uma chave artificial (ID).

|  |  |  |
| --- | --- | --- |
| ID (atributo chave inserido artificialmente) | Municípios | Estados |
| 1 | Bom Jesus | RN |
| 2 | Bom Jesus | RS |
| 3 | Bom Jesus | SC |
|  |  |  |

Essas chaves artificiais podem ser configuradas para ser automaticamente geradas pelo SGBD. Assim, para o caso anterior, ao se acrescentar novos cadastros, só precisaríamos informar o município e o estado e, assim, o SGBD automaticamente atribuiria um ID único para cada nova instância da relação.

Uma chave estrangeira é um atributo em uma relação que é utilizado para referenciar uma outra instância de alguma relação. Assim, um atributo chave estrangeira conterá o valor de uma chave primária de uma instância e outra relação. Exemplo: Um funcionário que trabalha para um departamento:

Departamento (Número, Nome, CPF-Gerente)

Funcionário (CPF, Nomes, Número\_Depto).

O esquema de BD relacional acima engloba 2 relações. A chave estrangeira aponta para a chave primária. Sempre vou ter Nomedarelação (atributos) = Esquema.

Os atributos sublinhados indicam as chaves primárias.

|  |
| --- |
| Departamento |
| Número | Nome | CPF-Gerente |
| 01 | RH | 124.482.396-14 |

|  |
| --- |
| Funcionário |
| CPF | Nome | Número-Depto |
| 124.482.396-14 | Vitória | 01 |

Assim, toda chave estrangeira se remete a algum valor de chave primária para alguma instância em outra relação (ou na própria relação também). Assim, através de chaves primárias e estrangeiras, conseguimos relacionar as relações em nosso esquema.

**Restrições do modo relacional** = Restrição de integridade das instâncias das relações. Impõe que nenhum valor da chave primária pode ser NULL (nulo, vazio), pois isso impediria a identificação de uma relação. **Restrição de integridade referencial** – Impõe a consistência no relacionamento entre as relações, isto é, que cada chave estrangeira em uma instância de relação possua um valor que, de fato, seja chave primária de alguma instância. Essa regra impediria por exemplo, que o atributo Numero-Depto de funcionário tivesse o valor 75, pos não existe instância na relação Departamento com este número. Em alguns casos, pode-se permitir que atributos chave estrangeira adquiram o valor NULL.

**Introdução a SQL:**

A SQL (Structured Query Language) é um padrão para SGBD’s relacionais, o que traz um bom nível de portabilidade entre diferentes SGBD’s. SQL possui instruções para a criação de um banco de dados, definição dos dados, consultas e atualizações como inserções, modificações e remoções. Assim, SQL pode funcionar como linguagem de definição de dados (DDL) e como linguagem de manipulação de dados (DML).

Definições e tipos de dados em SQL.

Os SGBD’s modernos implementam relações como tabelas. Assim, cada tupla (representante da relação) é vista como uma linha, e cada atributo é visto como uma coluna. Versões diferentes e mais recentes da SQL incluem o conceito de esquema, que pode ser entendido como uma “separação” do banco de dados de acordo com a necessidade de acesso e modificação dos dados. Um esquema SQL é identificado por um nome de esquema, identificadores de autorização, que indicam usuários ou proprietários do esquema, descritores para cada elemento do esquema, inclusive tabelas, restrições, views, domínios e outras construções.

**CREATE SCHEMA** schema\_name [**AUTHORIZATION** user\_name][schema\_element […]];

**CREATE SCHEMA** Projeto **AUTHORIZATION** ‘wendelmelo’**;**

Os comandos SQL não são sensíveis a caixa (pode-se usar letras maiúsculas ou minúsculas). As linhas escritas em SQL devem terminar com ponto e vírgula.

**CREATE SCHEMA** graduação;

Um catálogo é uma coleção de esquemas em um ambiente SQL. No nosso exemplo, os esquemas projeto e graduação estão em nosso catálogo. Além dos esquemas criados pelo usuário, por default, há também o esquema public.

SQL trabalha com a ideia de esquema corrente. Assim, os comandos de definição de dados serão aplicados ao sistema corrente. Para selecionar qual esquema será adotado como sendo corrente, podemos fazer:

**SET** search-path **TO** [nomedoesquema];

Por exemplo:

**SET** search\_path **TO** Projeto; --Muda para o esquema Projeto.

Se o search\_path **não** for setado para nenhum esquema específico, o esquema público será atualizado. Adicionalmente, cada catálogo contém um esquema especial denominado **INFORMATION\_SCHEMA**, que contém informações sobre todos os demais esquemas no catálogo.

Tipos de dados de atributo e domínio em SQL:

1. Numéricos: (1.1) Números inteiros: Integer, int ou small int; (1.2) Ponto Flutuante: Float, real ou double precision. O formato dos números pode ser declarado usando decimal (i, j) ou dec (i, j) ou Numeric (i, j) onde i é o número máximo de dígitos e j é o número próximo de dígitos para a parte decimal.
2. Cadeia de caracteres. (2.1) De tamanho fixo: Char (n) ou Character (n); (2.2) De tamanho variável: Varchar (n) ou Char Varying (n) ou Character Varying (n). Outros tipos são Character Large Object ou CLOB, onde se pode especificar o tamanho máximo em Kilobytes (K), Megabytes (M) ou Gigabytes (G): CLOB (ISM) – (ISMegabytes de caracteres).
3. Booleano: Além dos valores TRUE E FALSE, há também o valor UNKNOWN, devido aos valores NULL

Aula 13/10

**Tipos de dados (continuação)**

* Cadeia de bits: Podem ser de tamanho fixo BIT(n) pi de tamanho variável BIT VARYING(n). Ex: B'10110'. Outro tipo de cadeia de bit é o BINARY LARGE OBJECT ou BLOB para grandes cadeias de bits: BLOB(10k) (10 kilobits), BLOB(10M) (10 Megabyts), BLOC(10G) (10 Gigabytes). BLOB's são úteis para dados binários como imagens ou vídeo aos mesmos arquivos em geral.
* TIME: Representa uma hora do dia no formato HH:MM:SS. Ex: TIME: '07:45:34'. Há ainda o tipo de TIME(i) que especifica i casas decimais para os segundos (para medidas de tempo mais precisas). Há também o tipo TIME WITH TIME ZONE para especificar o deslocamento com base no fuso horário universal padrão na faixa de +13:00 a –12:59.
* DATE: Representa uma data válida, por exemplo na forma DD-MM-YYYY. Ex: DATE'10-03-1991'
* TIMESTAMP ou DATE/TIME representa uma data junto com uma hora. Obs: pode-se utilizar operadores de comparação (<, <=, >, >=) para comparar datas e horas.
* INTERVAL: Intervalor-valor relativo que pode ser usado para incrementar ou decrementar um valor absoluto de uma data/hora. Os intervalos são qualificados para serem ANO/MÊS ou DIA/HORA.

Em algumas interpretações de SQL, podemos criar domínios para atributos.

CREATE DOMAIN TIPO\_CPF AS CHAR(11)

Por exemplo, como um typedef na linguagem C. Se tivermos muitas tabelas armazenando CPF, e um dia CPF's passassem a ter 12 dígitos, será mais fácil realizar a atualização mudando apenas a especificação do domínio em vez de cada atributo CPF em cada tabela.

Podemos utilizar a cláusula CHECK para especificar restrições sobre um domínio. Por exemplo, suponhamos que desejamos especificar um domínio para um atributo que armazena a altura de uma pessoa:

CREATE DOMAIN TIPO\_ALTURA AS FLOAT;

CHECK(TIPO\_ALTURA > 0 AND TIPO\_ALTURA <3);

Criação de tabelas em SQL (www.postgresql.org)

SGBD's modernos implementam relações (entidades) por meio de tabelas. Podemos então criar uma tabela para representar a relação por meio do comando:

CREATE TABLE <nome da tabela> (

{ <nome da coluna> <tipo coluna [<restrição atributo>],}^t

{ <restrição tabela>}^t

|  |
| --- |
| contato |
| - Id: int (PK) |
| - Nome: varchar (200) |
| - dt\_nasc: date |
| - telefone: D\_TEL |

CREATE DOMAIN D\_TEL AS VARCHAR(14);

CREATE TABLE contato(

Id INT SERIAL,

nome VARCHAR(200) NOT NULL,

dt\_nasc DATE,

telefone D\_TEL,

PRIMARY KEY (id));

**Aula 17/10/17**

*Gerando a tabela Projeto*

1. CREATE TABLE Projeto (
2. id INT SERIAL,
3. nome VARCHAR(500) UNIQUE,
4. dt\_inicio DATE NOT NULL,
5. dt\_fim DATE NOT NULL,
6. interno BOOLEAN DEFAULT TRUE,
7. coordenador INT,
8. PRIMARY KEY(id)
9. FOREIGN KEY(id),
10. FOREIGN KEY(coordenador) REFERENCES contato(id),
11. CHECK(dt\_fim >= dt\_inicio));

|  |
| --- |
| **Contato** |
| **- id: int (PK)** |
| **- nome: varchar (200)** |
| **- dt\_nasc: date** |
| **- telefone: D\_TEL** |

 1

 n

|  |
| --- |
| **Projeto** |
| **- id: int (PK)** |
| **- nome: varchar (500)** |
| **- dt\_inicio: date** |
| **- dt\_fim: date** |
| **- interno: boolean** |
| **- coordenador: (FK\_contato)** |

 A linha 10 coloca uma restrição sobre qualquer tupla a ser inserida na tabela Projeto. É preciso tomar cuidado! Aqui, a lógica booleana é de 3 valores: TRUE, FALSE e UNKNOWN. Portanto, se dt\_fim ou dt\_inicio pudessem estar vazios (isto é, com valor NULL), nestes casos o resultado da comparação saia UNKNOWN e não passaria no CHECK. Observe que, pelas linhas 4 e 5 especificamos que dt\_inicio e dt\_fim não podem estar vazios.

A partir dos comandos SQL que cria a tabela Projeto, podemos observar:

* A cláusula SERIAL (linha 2) especifica que o respectivo campo é de incremento automático. Assim, podemos inserir tuplas na tabela e o próprio SGBD será capaz de gerar esse valor.
* A cláusula UNIQUE (linha 3) serve para especificar que um campo que não é chave primária (nome) possui valor único para cada tupla (chave candidata). Assim, se impõe que não podem haver dois projetos com o mesmo nome.
* A cláusula NOT NULL (linhas 4 e 5) especifica que os respectivos campos (dt\_inicio e dt\_fim) não podem estar fazias.
* A cláusula DEFAULT <valor> (linhas 6) especifica um valor padrão para o respectivo campo (inteiros), para o caso do mesmo não ser fornecido na inserção de uma tupla na tabela.
* A cláusula PRIMARY KEY (<campo>) (linha 8) especifica os campos que compõem a chave primária.
* A cláusula FOREIGN KEY (<campo>) REFERENCES <tabela> (<campos>) (linha 9) especifica chance estrangeira (coordenador).

Pode-se ainda nomear uma restrição com a cláusula CONSTRAINT. Por exemplo, poderíamos substituir a linha 8 por

1. CONTRAINT chaveProjeto PRIMARY KEY (id)

Nomeando restrições, é possível removê-las ou substituí-las posteriormente.

Tratando a integridade referencial

Ao se especificar uma chave estrangeira, podemos especificar também como o SGBD lidará com a integridade referencial. No nosso exemplo, se uma operação de alteração ou remoção de tupla na tabela Contato violar a integridade referencial da tabela Projeto (isto é, deixar algum projeto com valor inválido para o campo coordenador), a ação padrão do SGBD será rejeitar a respectiva atualização. A ação padrão do SGBD é rejeitar qualquer operação que possa ferir essa integridade.

**Aula 20/10/17**

|  |
| --- |
| **Primeiro Trabalho**Entregar comandos SQL para a criação das tabelas do último exercício (hangar de aviões) |

Podemos modificar este comportamento através das cláusulas:

* ON DELETE <ação>: para especificar ação no caso de remoção
* ON UPDATE <ação>: para especificar ação no caso de atualização

As possíveis ações são:

* SET NULL: mudar campo chave estrangeira para NULL
* SET DEFAULT: mudar campo chave estrangeira para valor default
* CASCADE: propagar a operação na tabela em questão

Exemplo:

|  |
| --- |
| 1. CREATE TABLE Projeto (
2. id INT SERIAL,
3. nome VARCHAR(500) UNIQUE,
4. dt\_inicio DATE NOT NULL,
5. dt\_fim DATE NOT NULL,
6. interno BOOLEAN DEFAULT TRUE,
7. coordenador INT
8. PRIMARY KEY(id),
9. FOREIGN KEY(coordenador) REFERENCES contato(id) ON DELETE SET NULL ON UPDATE CASCADE);
 |

Obs.: Se usarmos ON DELETE CASCADE, quando um contato for deletado, todos os seus projetos serão deletados também.

Inserindo dados em uma tabela

A cláusula INSERT nos permite inserir tuplas (linhas) em uma tabela:

|  |  |  |
| --- | --- | --- |
| Criando primeiro a tabela pessoa:1. CREATE TABLE Pessoa (
2. id INT SERIAL,
3. nome VARCHAR(200),
4. dt\_nasc DATE,
5. sexo CHAR(1),
6. PRIMARY KEY(id));
 |  |  Pessoa* id: int
* nome: varchar(200)
* dt\_nasc: date
* sexo: char(1)
 |
|  |  |  |
| Primeira formaDesta forma, os atributos são inseridos na ordem que está especificado na tabela.1. INSERT INTO <tabela> VALUES <linhas>;

Exemplo:1. INSERT INTO pessoa VALUES

(1, ‘Jessica Gomes’, ’19-03-1991’, ‘F’), (2, ‘Lucas Sacci, ’27-06-1990’, ‘M’), (3, ‘Raquel Maia’, ’29-04-1987’, ‘F’); |

|  |
| --- |
| Segunda formaDesta forma, especificamos os atributos a serem inseridos.1. INSERT INTO <tabela>(variável 1, variável 3, variável 8) VALUES <linhas>;

Exemplo:1. INSERT INTO pessoa(dt\_nasc, nome) VALUES

(’09-10-1989’,‘Renata Campos’);  |

**Aula 24/10/17**

**Deletando linhas de uma tabela**

O comando DELETE remove tuplas de uma de uma tabela. Pode ser incluída uma cláusula WHERE para especificar quais tuplas devem ser excluídas:

|  |
| --- |
| Forma geral:DELETE FROM <tabela> [WHERE <condição de seleção>];Exemplos:DELETE FROM Pessoa WHERE id=3;DELETE FROM Pessoa WHERE sexo=’M’;DELETE FROM Pessoa WHERE sexo IS NULL;DELETE FROM Pessoa WHERE sexo IS NOT NULL; |

PS.: podem ser apagados nenhuma linha da tabela, quando nenhum caso atender a condição *WHERE id=3,* um caso quando apenas um caso atender a condição *WHERE id=3,* e todos os casos (a tabela inteira) quando todos os casos atenderam a condição *WHERE id=3.*

Pode-se utilizar os operadores lógicos na condição de seleção.

|  |
| --- |
| DELETE FROM Pessoa WHERE sexo=’F’ and dt\_nasc <=’01-01-1970’; |

**Atualização as linhas de uma tabela**

O comando UPDATE atualiza tuplas de uma tabela. Assim como DELETE, UPDATE aceita uma cláusula WHERE para especificar quais linhas devem ser atualizadas:

|  |
| --- |
| Forma geral:UPDATE <tabela> SET {<coluna> = <expressão>} [WHERE <condição de seleção>];Exemplo:UPDATE Pessoa SET nome = ‘Jéssica Melo’ WHERE id = 1;UPDATE Pessoa SET sexo = ‘F’, nome=’Lara Sacci’ WHERE id = 2;UPDATE Pessoa SET sexo=NULL WHERE sexo=’M’; |

*Atenção: um erro comum, principalmente para principiantes, é esquecer a cláusula WHERE ao se utilizar DELETE e UPDATE:*

|  |
| --- |
| *DELETE from Pessoa;**UPDATE Pessoa SET sexo=’M’;* |

**Alterando a definição de uma tabela**

Podemos usar o comando ALTER TABLE para alterar a definição de uma tabela:

* Adicionando uma coluna:

|  |
| --- |
| ALTER TABLE <tabela> ADD COLUMN <coluna> <tipo>;Exemplo:ALTER TABLE Pessoa ADD COLUMN salario float; |

* Removendo uma coluna

|  |
| --- |
| ALTER TABLE <tabela> DROP COLUMN <coluna> <tipo de remoção>;Exemplo: ALTER TABLE Pessoa DROP COLUMN sexo RESTRICT; |

Onde <tipo de remoção> pode ser CASCADE ou RESTRICT.

**Aula 31/10/17**

|  |
| --- |
| Atividade para o dia da prova - 17/11Fazer a representação gráfica do código no site do professor (banco de dados de cargas aéreas) |

**Aula 03/11/17**

O comando select nos permite acessar (consultar) os dados armazenados nas tabelas do banco de dados.

**SELECT <lista de atributos que quero resgatar> FROM <lista de tabelas> WHERE <condição que tem que ser atendida para as linhas entrarem no resultado>;**

Exemplo: SELECT CodEmp, Nome FROM Empresa;

SELECT Nome, CodEmp FROM Empresa;

SELECT e.Nome, e.CodEmp FROM Empresa e; --O e na frente de empresa é um apelido. Uso quando vo misturar várias tabelas no SELECT.

SELECT e.Nome, a.Tipo FROM Empresa e, Aviao a; --Faz o produto cartesiano de cada linha da tabela avião com as linhas da tabela empresa. Associou BOING2 com todas as empresas, sendo que ele é um avião da GOL.

SELECT \* From Viagem; --Quero \* todas as colunas e linhas de uma determinada tabela.

SELECT \* From Empresa;

Podemos especificar ordenação com ORDER BY.

Select \* FROM Empresa ORDER BY CodEMP ASC; -- ASC de ascendente, ordem alfabética. Posso usar o DESC, que é o contrário. Funciona para qualquer tipo que seja ordenável.

SELECT CodEmp FROM Voo;

A cláusula DISTINCT seleciona tuplas ou linhas distintas no resultado.

SELECT DISTINCT CodEmp FROM Voo; --Resultado só tem tuplas distntas. Elimina valores repetidos.

SELECT DISTINCT CodEmp, CodVoo FROM Voo; --Aplicasse o distinct no resultado todo.

A cláusula WHERE nos permite especificar condições a serem atendidas pelas linhas retornadas.

SELECT \* FROM Voo WHERE CodEmp = 'AZU'; --Agora especifiquei que quero as linhas da tabela Voo onde o código da empresa seja AZUL.

SELECT \* FROM Voo WHERE CodEmp = 'TAM';

SELECT \* FROM Voo WHERE CodEmp = 'TAM' OR CodEmp = 'AZU';

SELECT \* FROM Voo WHERE Codemp <> 'GLO' -- <> especifico que o código da empresa seja DIFERENTE de GOL.

Quero selecionar as rotas de vôos após o meio dias.

SELECT \* FROM Rota WHERE H\_saida > '12:00:00'; -Consigo fazer esse tipo de operação porque o campo é do tipo TIME.

SELECT \* FROM Rota WHERE H\_saida > '12:00:00' ORDER BY H\_saida ASC;

Posso dar select em mais de uma tabela. Vou dar select na tabela Voo e na tabela Empresa.

SELECT \* FROM Voo, Empresa; --A tabela Voo só tem 13 linhas e aparecem 39 resultados. Por que pegou cada uma das 13 de Voo e casou com as 3da tabela Empresa. O SELECT faz esse produto cartesiano por padrão. Nessa hora, entra o relacionamento entre as tabelas. Ambas tem a coluna chamada CodEmp (é a chave estrageira na tabela Voo). Vamos especificar este relacionamento na cláusula WHERE.

SELECT \* FROM Voo v, Empresa e WHERE v.CodEmp = e.CodEmp. --Só quero resultados onde as duas linhas forem iguais. Só me mostra os 13 resultados. Casou certinho as tabelas.

SELECT v.codvoo, v.codEmp, e.Nome FROM Voo v, Empresa e WHERE v.codEmp = e.CodEmp;

Quando uso o WHERE especifico condições.

INSERT INTO Empresa(CodEmp, Nome) VALUES ('AVI', 'Viação Avianca');

SELECT \* FROM Empresa;

UPDATE Empresa SET CodEmp = 'Ava', nome = 'Transportes Avianca S.A.'; --Altera TODOS para AVA. Preciso do WHERE para dar um update. Não me deixa alterar o que for chave primária.

UPDATE Empresa SET nome = 'Transportes Avianca S.A.' WHERE codemp = 'AVI';

DELETE FROM empresa; --Deleta todas as linhas da tabela empresa.

DELETE FROM empresa WHERE CodEmp = 'AVI'; --Deleta as linhas que contenham código da empresa AVI. Como codemp era chave primária, só deletou uma linha (máximo).

Vôos que saíram com +10min de atraso.

SELECT FROM Viagem v, Rota WHERE v.h\_Saida\_real – r.h\_saida> '00:10:00' --Só tem horário de saída daquela viagem. O que deveria ter saído é rota. Preciso acessar as duas. --Cruzou tudo.

SELECT FROM Viagem v, Rota WHERE v.h\_Saida\_real – r.h\_saida> '00:10:00' AND v.codRota = r.codRota;

**Aula 07/11/17**

**Casamento de padrões com o comando LIKE:**

A cláusula like permite casar padrões em uma string. São utilizados dois caracteres especiais reservados:

|  |
| --- |
| O caractere % casa com qualquer sequência de 0 ou mais caracteresO caractere \_ casa um único caracterePor exemplo, todas as linhas da tabela avião cujo tipo comece com BOEING + 1 caractere qualquer:SELECT \* FROM Aviao WHERE Tipo LIKE ‘BOEING\_’;Exemplo, todas as viagens feitas no mês de agosto:SELECT \* FROM Viagem v WHERE to\_char(v.Data, ‘DD/MM/YYYY’) LIKE ‘\_\_/08/\_\_\_\_’;Viagens feitas no mês de Agosto de 2017:SELECT \* FROM Viagem v WHERE v.Data >= ‘01/08/2017’ AND v.Data <= ‘31/08/2017’;  |

//////////// Ele explicou do comando between mas não consegui copiar ☹ ////////////

Operações em conjunto

UNION: união

INTERSEPT: interseção

EXCEPT: diferença

Para fazer as operações com conjuntos corretamente, as tabelas resultantes devem ser compatíveis

|  |
| --- |
| SELECT nome FROM empresaUNIONSELECT tipo FROM aviaoExemplo: Listando todos os aeroportos que são origens ou destinos das rotas:SELECT DISTINCT origem FROM rotaUNIONSELECT DISTINCT destino FROM rota; |

**Aula 10/11/17**

**Union –** Serve para fazer união de conjuntos. Distinct – Seleciona resultados distintos.

Set search\_path to cargaraereas; --Para usar o caminho certo.

Select Distinct origem from Rota --Não repete resultados. Estou listando todas as origens que vem da tabela rota.

UNION

Select Distinct destino from Rota; –Vou ter todos os aeroportos que são usados como origem e os que são usados como destino. --Union é uma operação de conjunto. Quando uso, exclui automaticamente resultados iguais. Poderia até apagar o distinct que o resultado seria o mesmo por conta da filtragem de union. Tenho que unir tabelas com o mesmo número de colunas e as colunas devem estar na mesma ordem. Partindo deste select, posso criar uma tabela de aeroportos, que inclui apenas os aeroportos de origem e destino.

--criando tabela para armazenas aeroportos da rota

CREATE TABLE Aeroporto(

Sigla varchar(3),

Nome varchar(500),

Endereco varchar(1000),

Primary key (Sigla)

);

--Povoando a tabela Aeroporto com os resultados do select anterior

INSERT INTO Aeroporto (Sigla) --Especifico quais são os atributos que eu quero inserir na tabela, eu só quero inserir na coluna sigla, os resultados que vem dessa sql aqui:

Select distinct origem FROM rota

UNION

Select distinct destino FROM rota;

Select \* from aeroporto; --Vou ver que tenho exatamente os resultados que vieram dos select e dos unions que a gente usou. Nome e endereço ficaram com valor NULL.

Select \* from aeroporto WHERE nome = NULL; --Otendo linhas de aeroporto cujo campo nome está vazio. Note o uso do operador IS NULL. Vai retornar falso. Para ele, um nulo nunca é igual a outro. Se o campo está nulo, ele está em branco.

Select \* from aeroporto WHERE nome = IS NOT NULL; --Obtendo linhas cujo nome não está vazio.

UPDATE Aeroporto SET endereco = 'XXX' WHERE endereco IS NULL -Preenchendo o campo endereço na tabela aeroporto com o valor 'XXX' para as linhas onde esse campo estiver vazio. Sem o where, ele atualizaria todas.

--A lógica na SQL possui 3 valores: TRUE (verdadeiro), False (falso) e Unknown (desconhecido).

--Operadores In e Not In: Permitem comparação em uma lista de valores.

--Listando as rotas que partem dos aeroportos 'SAO', 'GIG' e 'BHZ':

Select \* from rota r WHERE r.origem = 'SÃO' OR r.origem = 'GIG' OR r.origem = 'BHZ';

--Podemos usar o operador IN:

SELECT \* FROM rota r WHERE r.origem IN ('SÃO', 'GIG', 'BHZ');

--Listando rotas que não partem desses aeroportos:

SELECT \* FROM rota r WHERE r.origem NOT IN ('SÃO', 'GIG', 'BHZ');

SELECT \* FROM rota r WHERE r.origem <> 'SÃO' AND r.origem <> 'GIG' AND r.origem <> 'BHZ';

--Podemos dar select simultaneamente em duas tabelas:

SELECT \* FROM empresa e, voo v WHERE e.codEmp = v.codEmp;

SELECT v.codVoo, e.codEmp, e.nome FROM empresa e, voo v WHERE e.codEmp = v.codEmp; --Se eu só desse um select na tabela voo, não apareceria empresa. Aqui eu consigo pegar código do voo, código da empresa e o nome da empresa.

--Essa mesma consulta pode ser realizada a partir do conceito de junção interna de tabelas (JOIN ou INNER JOIN). É como se eu estivesse colando uma tabela do lado da outra.

SELECT \* FROM Empresa e JOIN voo v ON e.codEmp = v.codEmp; --O on especifica o relacionamento das duas tabelas envolvidas na junção.

--No caso acima, pode-se usar junção natural (Natural join). Nesse caso, o relacionamento é feito usando todos os atributos de mesmo nome nas duas tabelas:

 SELECT \* FROM Empresa e NATURAL JOIN voo v; --é melhor usar JOIN mesmo. Se eu usar o NATURAL JOIN e tiver um atributo com o mesmo nome nas 2 tabelas, que não faça parte do relacionamento, posso ter atributos errados.

--É preciso ter cuidado com NATURAL JOIN pois ele usará TODOS os atributos de mesmo nome nas duas tabelas para a junção. Assim, se houver acidentalmente dois atributos de mesmo nome nas tabelas que faça parte do relacionamento, ele será indevidamente utilizado. Se não houver atributos de mesmo nome, será feito o produto cartesiano normal das duas tabelas. Entretanto, eu sempre tenho a opção de renomear atributos. Pode-se renomear a coluna recuperada com select:

 SELECT e.codEMp as Codigo\_empresa FROM Empresa e Natural JOIN voo v;

--O conceito de junção interna combina apenas as tuplas das relações onde o relacionamento é atendido. Pode ocorrer de alguma tupla em uma relação não combinar com qualquer tupla da outra relação e, ainda assim, precisar ser listada. Nesse caso, é preciso recorrer ao conceito de junção externa (outer join).

--Exemplo: vamos inserir uma nova empresa:

INSERT INTO Empresa VALUES ('AVI', 'Viação Avianca');

--Observe que a empresa inserida não possui voo relacionado a ela, o que faria com que não fosse listada em uma junção interna.

SELECT \* FROM Empresa Natural Join Voo;

--Para que a nova empresa fosse listada, seria preciso uma junção externa à esquerda (Left Outer Join). Significa que as linhas da tabela empresa que não estão ligadas a ninguém em vôo, tem que ser listadas.

SELECT \* FROM Empresa e Left Outer Join Voo v On e.codEmp = v.codEmp;

Se quiséssemos listar as linhas sem relacionamento da tabela Voo, usar[iamos junção extterna a direita (Right Outer Join). Para listar linhas sem relacionamento de ambas as tabelas, usaríamos junção externa completa (Full outer join).

--Pode-se usar junções envolvendo mais de dias tabelas:

Select \* From empresa e JOIN voo v ON e.codEmp = v.codEmp JOIN rota r ON r.codVoo = v.codVoo;

P1: Tudo que ele deu até antes do SELECT.

**Aula 14/11/17 – Tô fazendo anotações coloridas pq a anotação imita a vida e a vida tem que ser colorida!!!!!!!!!!!!!!!!!!!!!!**

|  |
| --- |
| QUESTÃO DE PROVA: O que o comando SELECT retorna? O SELECT retorna uma tabela. |

**Funções de agregação**

Resumem informação de várias linhas em uma única linha. As funções mais comuns são:

|  |
| --- |
| COUNT: contagem de linhasSUM: soma de valores em uma colunaMIN: valor mínimo em uma colunaMAX: valor máximo em uma colunaAVG: média de uma coluna |

Exemplo: contar quantas linhas

|  |
| --- |
| SELECT COUNT(\*) FROM Viagem; |

Podemos renomear a coluna resultante para numViagens:

|  |
| --- |
| SELECT COUNT(\*) AS numViagem FROM Viagem; |

Podemos contar o número de resultados em uma coluna. Neste código abaixo, o comando só contará as linhas que estiverem preenchidas na coluna. Valores nulos serão ignorados:

|  |
| --- |
| SELECT COUNT(h\_saida\_real) FROM Viagem; |

Podemos usar a cláusula WHERE para, por exemplo, contar os voos da azul:

|  |
| --- |
| SELECT COUNT(\*) FROM Voo WHERE codEmp = ‘AZU’; |

A cláusula DISTINCT pode ser utilizada para contar valores distintos em uma coluna contando o número de tipos distintos de avião:

|  |
| --- |
| SELECT COUNT(DISTINCT tipo) FROM Aviao; |

Soma das capacidades de todos os aviões da TAM:

|  |
| --- |
| SELECT SUM(capacidade) FROM Aviao WHERE codEmp = ‘TAM’; |

A rota que chega mais cedo ao aeroporto GIG:

|  |
| --- |
| SELECT MIN(h\_chegada FROM Rota WHERE destino = ‘GIG’; |

A data da última viagem registrada:

|  |
| --- |
| SELECT MAX(v.data) FROM Viagem v; |

A média dos atrasos das viagens que tiveram atraso. Faremos isto em duas partes:

A primeira parte é programarmos uma tabela de atrasos. Este comando gera uma tabela onde o resultado é uma operação matemática.

|  |
| --- |
| SELECT (v.h\_saida\_real – r.h\_saida)  FROM Viagem v JOIN Rota r ON v.codRota = r.codRota WHERE v.h\_saida\_real > r.h\_saida; |

A segunda parte é retirarmos a média da coluna com o comando AVG no mesmo comando:

|  |
| --- |
| SELECT AVG(v.h\_saida\_real – r.h\_saida)  FROM Viagem v JOIN Rota r ON v.codRota = r.codRota WHERE v.h\_saida\_real > r.h\_saida; |

Podemos usar várias funções de agregação juntas em relação a capacidade dos aviões:

|  |
| --- |
| SELECT COUNT(\*), SUM(capacidade), MIN(capacidade), MAX(capacidade),  AVG(capacidade) FROM Aviao; |

Vamos adiantar um pouco a matéria e usar um SELECT aninhado para listar tabela avião com a média da capacidade ao lado:

|  |
| --- |
| SELECT \* FROM Aviao, (SELECT AVG(capacidade) FROM Aviao) m; |

O que é preciso entender é que podemos encarar o resultado em um SELECT como uma tabela. Se o resultado SELECT é uma tabela, podemos coloca-lo em uma cláusula FROM. Assim, o SELECT acima deve fazer o produto cartesiano entre as tabelas Avião e a tabela retornada por esse SELECT:

|  |
| --- |
| SELECT \* FROM Aviao, (SELECT AVG(capacidade) FROM Aviao) m; |

Assim, podemos listar apenas os aviões cuja capacidade está acima da média:

|  |
| --- |
| SELECT \* FROM Aviao a, (SELECT AVG(capacidade) FROM Aviao) m WHERE a.capacidade >= m.avg; |

A agregação com agrupamentos: podemos aplicar as funções de agregação a subgrupos de tuplas em uma relação, onde os subgrupos são baseados em valores de atributos. Para especificar esses agrupamentos, podemos usar as cláusulas GROUP BY e HAVING. GROUP BY especifica os atributos de agrupamento, que também devem aparecer na cláusula SELECT.

Contando o número de aviões de cada empresa

|  |
| --- |
| SELECT codEmp, COUNT(\*) FROM Aviao GROUP BY codEmp; |

Contando o número de aviões junto com o total

|  |
| --- |
| SELECT codEmp, COUNT(\*) FROM Aviao GROUP BY codEmpUNIONSELECT ‘Total’, COUNT(\*) FROM Aviao; |

A cláusula HAVING permite especificar condições que cada grupo (como um todo) deve satisfazer para ser listado no resultado. Essas condições devem ser construídas sobre alguma(s) da(s) função(ões) de agregação utilizada(s).

Listando empresas que tenham mais que dois aviões:

|  |
| --- |
| SELECT codEmp, COUNT(\*) FROM Aviao GROUP BY codEmp HAVING COUNT(\*) > 2; |

Note que a cláusula HAVING se destina a especificar condições para escolher grupos inteiros. Não confundir com WHERE, onde especificamos condições para as linhas.

Contando o número de rotas de cada empresa:

|  |
| --- |
| SELECT v.codEmp, COUNT(\*) FROM Voo v, Rota r WHERE v.codVoo = r.codVoo GROUP BY codEmp; |

Listando as empresas com mais de 7 rotas:

|  |
| --- |
| SELECT v.codEmp, COUNT(\*) FROM Voo v, Rota r WHERE v.codVoo = r.codVoo GROUP BY codEmp HAVING COUNT(\*) > 7; |

|  |
| --- |
| SELECT codEmp, COUNT(\*) FROM Aviao GROUP BY codEmp; |

|  |
| --- |
| SELECT codEmp, COUNT(\*) FROM Aviao GROUP BY codEmp; |

|  |
| --- |
| SELECT codEmp, COUNT(\*) FROM Aviao GROUP BY codEmp; |

**Aula 17/11/17**

**Visões**

No contexto de SQL, uma visão (view) é uma tabela derivada de outras tabelas. Essas outras tabelas podem ser tabelas da base ou outras views. Views também são denominadas como tabelas virtuais, pois não necessariamente estão armazenadas em forma física no BD. Em geral, pode-se fazer qualquer consulta com SELECT sobre uma view como se ela fosse qualquer tabela da base. No entanto, as operações que envolvem atualização dos dados ficam mais restritas.

Exemplo: vamos listas as viagens juntamente com os nomes das empresas responsáveis.

Primeiro: criando a View

|  |
| --- |
| CREATE VIEW TabelaoAS SELECT v.\*, e.\* FROM (((Viagem v JOIN Rota r ON v.codRota = r.codRota) JOIN Voo p ON r.codVoo = p.codVoo) JOIN Empresa e ON p.codEmp = e.codEmp); |

Consultando a View

|  |
| --- |
| SELECT \* FROM Tabelao; |

Criando uma view com funções de agregação:

|  |
| --- |
| CREATE VIEW EmpVoo (CodigoEmp, NomeEmpresa, NumeroVoos)AS SELECT e.codEmp, e.Nome, COUNT(v.codVoo) FROM Empresa e LEFT OUTER JOIN Voo v ON e.codEmp = v.codEmp GROUP BY e.codEmp, e.Nome; |

Consultando a View:

|  |
| --- |
| SELECT \* FROM EmpVoo; |

O SGBD é responsável por manter a view sempre atualizada, o que significa que mudanças nas tabelas que originaram a view devem se refletir na view.

|  |
| --- |
| SELECT \* FROM Tabelao; |

Iremos acrescentar uma empresa na tabela empresa para vermos a mudança automática na view. Atenção: iremos alterar somente a tabela empresa. A view se atualizará automaticamente.

|  |
| --- |
| INSERT INTO Empresa VALUES ('PTB', 'Passaredo Linhas Aéreas'); |

Ao consultarmos a View, veremos que a Passaredo foi inserida também na view.

|  |
| --- |
| SELECT \* FROM empVoo; |

Inserindo um vôo para a nova empresa.

|  |
| --- |
| INSERT INTO Voo VALUES (‘XXX’, ‘PTB’); |

Podemos ter problemas para atualizar a view. Por exemplo, vamos tentar atualizar o número de voos da avianca na view EmpVoo:

|  |
| --- |
| UPDATE EmpVoo SET numerovoos = 10 WHERE codigoemp = ‘AVI’; |

Algumas atualizações podem ser feitas (depende do banco):

|  |
| --- |
| UPDATE tabelao SET nome = ‘Avianca Linhas Aéreas’ WHERE codemp = ‘AVI’; |

Para deletar a view, podemos deletar usando o comando DROP VIEW:

|  |
| --- |
| DROP VIEW Tabelao; |

Consultas aninhadas. Algumas consultas precisam que valores existentes no banco de dados sejam buscados para, então, serem utilizados para compor uma coluna do resultado ou em uma condição de comparação. Podemos formular essas consultas usando SELECTS aninhados, que nada mais são do SELECTS incluídos dentro do contexto de outro SELECT:

|  |
| --- |
| SELECT \* FROM Tabelao; |

Primeiro caso: consultas que precisam apenas de um valor escalar (um número simples) para compor uma coluna do resultado ou para uso em uma comparação.

Podemos lembrar que o select permite que se retorne constantes (ou resultados de expressões) junto com colunas de tabela:

|  |
| --- |
| SELECT t.\*, 4 FROM Trecho t;SELECT 7\*3, t.\* FROM Trecho t; |

Uma vez que podemos listas valores constantes na tabela resultante, podemos então especificar que esse valor constante seja resultado de outro SELECT, desde que esse outro SELECT retorne um resultado com apenas uma linha e uma coluna:

Exemplo: vamos resgatar as linhas da tabela Trecho juntamente com a média das distâncias dos trechos:

|  |
| --- |
| SELECT t.\*, (SELECT AVG(distancia) FROM Trecho) FROM Trecho t; |

Podemos renomear a coluna vinda do SELECT

SELECT t.\*

**Aula 17/11/17**

Dependências Funcionais

Uma dependência funcionaç é uma restrição entre dois conjuntos de atributos X e Y, onde os valores de X determinam excclusivamente os valores de Y. Representamos a dependência funcional aqui descrita como:

$$X\rightarrow Y$$

Os valores dos atributos em Y são determinados pelos valores dos atributos em X (Y é funcionalmente dependente de X).

Em outras palavras a restrição anterior afirma que, para quaisquer duas tuplas (linhas) t1 e t2 de uma relação (tabela) que apresentam t1[x] = t2[x] (mesmo valor para X), necessariamente terão que apresentar t1[y] = t2[y] (mosmo valor para Y). Note que a rciproca, isto é, se t1[y] = t2[y] não necessariamente t1[x]=t2[x]

$$Cep\rightarrow \{Cidade, Estado\}$$

$$Logradouro, Cidade, Estado\rightarrow Cep?$$

$$Estado, Crea\rightarrow NomeEngenheiro$$

Note que, se X é uma chave candiata de uma relação R, então $X\rightarrow Y$ é verdadeiro para qualquer subconjunto Y de atributos de R.

$$CPF\rightarrow Nome, dataNasc, sexo$$

O uso principal das dependências funcionais é para descrever melhor um esquema de relação R ao especificar restrições sobre seus atributos que devem ser mantidas o tempo todo.

Notação do Diagrama:

Exemplo:

DF1: SSN-> Enome

DF2: Pnumero ->Pnome, Plocalização

DF3: SSN, Pnumero -> Horas

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SSN | Pnumero | Horas | Enome | Pnome | Plocalização |

Setinha DF1: de SSN para Enome

Setinha DF2: de Pnumero para Pnome, Plocalização

Setinha DF3: de SSN para Horas

Exemplo: Dada a seguinte relação

Cliente(idCliente, nome, telefone)

As seguintes relações estão corretas?

Nome -> telefone R:**nope**

Telefone-> nome R: **nope**

idCliente-> nome R: **yas!**

idCliente ->telefone R: **yupi!**

**Normalização**

Processo a partir do qual se aplicam regras à todas as relações de um esquema relacional para certificar que o mesmo satisfaz certa forma normal. As formas normais mais conhecidas são:

* 1ª forma normal (1FN)
* 2ª forma normal (2FN)
* 3ª forma normal (3FN)
* Forma normal de Boyce-Codd
* 4FN
* 5FN

Vantagens

* Melhor organização dos dados (sem misturardiferentes assuntos numa mesma relação desnecessariamente)
* Garante relações sem redundância desnecessária
* Tende a oferecer melhor recuperação dos dados

**Aula 12/12/17 (COMPLETAR)**

**Primeira forma normal:** Não pode ter atributo multivalorado

**Aula 15/12/17**

* **Atributo principal:** Atributo membro de alguma chave (mínimo) em uma relação.
* **Atributo não principal:** Atributo que não é membro de qualquer chave candidata (mínimo) em uma relação.

Uma dependência funcional X -> Y é uma dependência funcional se a remoção de qualquer atributo de X implicar que a dependência não se mantém mais. Por outro lado, se algum atributo de X puder ser removido e a dependência se mantiver, dizemos que X->Y é uma dependência funcional parcial.

Estado, CREA (DF total) -> Nome do engenheiro

CPF, RG -> Nome (DF Parcial)

Isto quer dizer que se retirarmos a chave CPF ou RG, o nome ainda poderia ser definido.

Segunda Forma Normal (2FN)

* Um esquema de relação R está na 2FN se cada atributo não principal em R for dependente funcional total da chave primária de R.

Exemplo: a relação a seguir não está na 2FN.

PedidoInfo(nro\_pedido, data, nro\_peça, descrição, qtd, preço)

PedidoInfo

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| nro\_pedido | data | nro\_peça | descricao | qtd | preço |
| 1 | 19\_03\_2017 | 301 | torneira | 2 | 200 |
| 1 | 19\_03\_2017 | 704 | martelo | 1 | 55 |
| 2 | 21\_03\_2017 | 704 | martelo | 3 | 60 |

Temos:

nro\_pedido -> data

nro\_peça -> descricao

Chave primária: {nro\_pedido, nro\_peça} -> {qtd, preço}

{qtd, preço} não são dependentes funcionais totais da chave. Isto, na segunda forma normal, é errado.

Em outras palavras, existem atributos que dependem da chave principal. Porque isto não pode? Por causa da redundância gerada.

A chave primária é composta por nro\_pedido é composta por nro\_peça. Temos os atributos data e descrição com dependência funcional parcial da chave, o que viola a segunda forma normal. Note que o fato de violar a segunda forma normal faz com que haja redundância de informação nos atributos data e descrição.

Para normalizar a relação para a 2FN, é neessário verificar se a chave primária é de fato mínima. Também pode ser necessário criar relações para os atributos que violam a normal agrupando-os por dependência, além do correto ajuste das chaves estrangeiras.

Pedido(nro\_pedido, data)

Pedido(nro\_peça, descrição)

Pedido(nro\_pedido(FK\_pedido), nro\_peça (FK\_peça), qtde, preço)

OBS: Se a chave primária de uma relação R não for artificial e só contiver um atributo, R estará na 2FN. (...)

**Terceira Forma Normal (3FN)**

Dependência Transitiva:

Uma dependência funcional X -> Y em R é dita transitiva se houver um conjunto de atirbutos Z que não seja chave candidata nem subconjunto da chave primária de R tal que X -> Z e Z -> T. (insto é, existem atributos não chave que são dependentes de outros abributos não chave.

Um esquema de relação R está na 3FN se:

* Está na 2FN;
* Nenhum atributo não principal de R for transitivamente dependente da chave.

No exemplo anterior com chave primária artificial não estaria na 3ª FN.