Programação Lógica

Bancos de dados relacionais em Prolog

Alexsandro Santos Soares
prof.asoares@gmail.com

16 de setembro de 2020

Bacharelado em Sistemas de Informação
Faculdade de Computação
Universidade Federal de Uberlândia

Sumário

Sumário i

Sumário ii

Introdução

Bancos de dados relacionais

Um banco de dados é uma coleção estruturada de dados armazenados.

Em um banco de dados relacional todos os dados estão agrupados conceitualmente em *relações*, que normalmente são visualizadas como tabelas.

Abaixo está a relação *Funcionário* e no próximo slide a relação *Departamento*.

Funcionário

matfunc	nome	numdepto	salário	matgerente
13	Marcelo	0	3000	19
21	Joana	1	2000	13
35	Bruno	1	2200	21
38	Wagner	1	1600	35
43	Sílvia	2	2500	13
61	Tiago	1	1700	21
89	Márcia	1	2300	35
42	Mirela	1	2000	35

Departamento

numdepto	nome	matgerente
1	Relações Públicas	21
2	Segurança	43

Uma coluna na tabela é chamada de atributo e possui um nome, por exemplo, `numdepto`. Todos os valores de um atributo pertecem a um domínio comum, por exemplo, cada salário pertence aos inteiros.

Uma relação é um conjunto de tuplas, também chamadas de registros e representadas por linhas da tabela, que consistem de valores para atributos. Por exemplo:

```
< 35, Bruno, 1, 2000, 21 >
```

Bancos de dados relacionais

As tuplas pertencem ao conjunto descrito por um esquema relacional que especifica nomes, domínios e ordem dos atributos, por exemplo:

Funcionário `< inteiro matfunc, string nome, inteiro numdepto, inteiro salário, inteiro matgerente >`

Departamento `< inteiro numdepto, string nome, inteiro matgerente >`

Uma relação pode ser alterada pela inserção, remoção ou atualização de alguma de suas tuplas. Estas operações são denominadas por manipulação de dados.

Uma consulta a um banco de dados é respondida pela enumeração das tuplas da relação resultante, ou pelo cálculo realizado por uma função de agregação sobre as tuplas , tal como a *média* ou o *total*.

Chave primária

Uma chave primária é uma combinação de um ou mais atributos que **nunca** se repetem na mesma tabela. Ou seja, a combinação é *única* na tabela e sem repetição.

Por ser única podemos usar uma chave primária para identificar de forma inequívoca uma determinada tupla podendo, assim, ser usada como índice.

Example

Na relação *Funcionário* o atributo *Matrícula do Funcionário* (matfunc) pode ser considerado uma chave primária.

Funcionário

matfunc	nome	numdepto	salário	matgerente
13	Marcelo	0	3000	19
21	Joana	1	2000	13
35	Bruno	1	2200	21
38	Wagner	1	1600	35

Chave primária

A chave primária pode ser:

simples se ela for formada por um único atributo da relação.

composta se ela for formada por mais de um atributo. Nesse caso, os valores individuais de cada atributo podem se repetir, mas nunca a combinação desses valores.

Um exemplo de chave composta seria a combinação do código de um livro e o código de um autor, em uma relação sobre livros.

Por convenção, costumamos sublinhar os atributos que são chaves primárias em uma relação. Por exemplo:

- Funcionário: matfunc, nome, numdepto, salário, matgerente
- Departamento: numdepto, nome, matgerente

Chave estrangeira

Denominamos de chave estrangeira um atributo de uma relação que for uma chave primária em outra relação.

Example

Na relação *Funcionário* tanto *numdepto* quanto *matgerente* são chaves estrangeiras, pois existirão outras relações nas quais essas chaves serão primárias permitindo, por exemplo, identificar o nome do gerente ou o nome do departamento.

Funcionário

matfunc	nome	numdepto	salário	matgerente
13	Marcelo	0	3000	19
21	Joana	1	2000	13
35	Bruno	1	2200	21
38	Wagner	1	1600	35

Operações primitivas

Operações primitivas: seleção

Muitas consultas podem ser expressas em função das seguintes operações primitivas sobre as relações:

Seleção escolhe tuplas para as quais uma dada condição é válida.

- Por exemplo, podemos selecionar de *Funcionário* aqueles funcionários do departamento 1 que ganham acima de 2100 (existem duas tuplas assim).

Funcionário

matfunc	nome	numdepto	salário	matgerente
13	Marcelo	0	3000	19
21	Joana	1	2000	13
35	Bruno	1	2200	21
38	Wagner	1	1600	35
43	Sílvia	2	2500	13
61	Tiago	1	1700	21
89	Márcia	1	2300	35
42	Mirela	1	2000	35

Operações primitivas: projeção

Projeção ignora alguns atributos com a reordenação opcional dos atributos restantes.

- Por exemplo, podemos projetar *Funcionário* sobre *nome*, *matfunc* e *salário* para obter

```
< Marcelo, 13, 3000 >
```

juntamente com mais outras sete tuplas.

nome	matfunc	salário
Marcelo	13	3000
Joana	21	2000
Bruno	35	2000
Wagner	38	1600
Sílvio	43	2400
Tiago	61	1700
Márcia	89	2100
Mirela	42	1700

Operações primitivas: junção

Junção de duas relações A e B forma uma nova relação consistindo das concatenações das tuplas de A com as de B, para as quais uma dada condição é válida.

- Por exemplo, a junção das relações *Funcionário* e *Departamento* tal que os números de departamento coincidam, consiste na tupla

```
<21, Joana, 1, 2000, 13, 1, Relações Públicas, 21>
```

juntamente com mais outras seis tuplas.

matfunc	nome	numdepto	salário	matger	numdepto	nome	matger
21	Joana	1	2000	13	1	Relações Públicas	21
35	Bruno	1	2200	21	1	Relações Públicas	21
38	Wagner	1	1600	35	1	Relações Públicas	21
43	Sílvia	2	2500	13	2	Segurança	43
61	Tiago	1	1700	21	1	Relações Públicas	21
89	Márcia	1	2300	35	1	Relações Públicas	21
42	Mirela	1	2000	35	1	Relações Públicas	21

Operações primitivas: produto

Produto é uma junção produto das relações *Funcionário* e *Departamento* consiste nas dezesseis tuplas abaixo.

matfunc	nome	numdepto	salário	matger	numdepto	nome	matger
13	Marcelo	0	3000	19	1	Relações Públicas	21
21	Joana	1	2000	13	1	Relações Públicas	21
35	Bruno	1	2200	21	1	Relações Públicas	21
38	Wagner	1	1600	35	1	Relações Públicas	21
43	Sílvia	2	2500	13	1	Relações Públicas	21
61	Tiago	1	1700	21	1	Relações Públicas	21
89	Márcia	1	2300	35	1	Relações Públicas	21
42	Mirela	1	2000	35	1	Relações Públicas	21
13	Marcelo	0	3000	19	2	Segurança	43
21	Joana	1	2000	13	2	Segurança	43
35	Bruno	1	2200	21	2	Segurança	43
38	Wagner	1	1600	35	2	Segurança	43
43	Sílvia	2	2500	13	2	Segurança	43
61	Tiago	1	1700	21	2	Segurança	43
89	Márcia	1	2300	35	2	Segurança	43
42	Mirela	1	2000	35	2	Segurança	43

Operações primitivas: operações sobre conjuntos

Operações sobre conjuntos, tais como a **união**, a **interseção** e a **diferença**, podem ser aplicadas a duas relações cujos atributos correspondentes pertençam ao mesmo domínio. Ou seja:

- Os esquemas das relações diferem somente nos nomes dados aos atributos.

Tradução em Prolog

Tradução das relações em Prolog

A terminologia de álgebra relacional pode ser traduzida em Prolog da seguinte forma:

Conceito	Tradução
Relação	Predicado
Atributo	Argumento do predicado
Tupla	Fato
Domínio	Valor

A relação *Funcionário* poderia ser expressa assim:

```
/* Funcionário */
/*    MatFunc, Nome, NumDepto, Salário, MatGerente */
:- dynamic funcionário/5.

funcionário(13, 'Marcelo', 0, 3000, 19).
funcionário(21, 'Joana',   1, 2000, 13).
/* demais tuplas da relação ... */
```

Manipulação de Dados

Três atividades são necessárias na manipulação de uma relação:

1. Adicionar um fato novo.
2. Remover um fato existente.
3. Atualizar um fato

A terceira atividade pode ser expressa em função das duas primeiras: remover o fato antigo e adicionar o novo.

Para adicionar um fato podemo usar `assert` e para remover `retract`, ou alguma das variações destes dois.

Tradução das operações primitivas

As operações primitivas sobre relações são expressas em função das chamadas de predicados.

Example

Gerar todas as tuplas para os funcionários do departamento 1 que ganhem acima de 2100.

```
seleciona(MatFunc, Nome, NumDepto, Salário, MatGerente):-
    funcionário(MatFunc, Nome, NumDepto, Salário, MatGerente),
    NumDepto = 1, Salário > 2100.
```

O predicado acima implementa uma *seleção*, cuja consulta poderia ser

```
?- seleciona(MatFunc, Nome, NumDepto, Salário, MatGerente),
   write( (MatFunc, Nome, NumDepto, Salário, MatGerente) ),
   nl, fail.

35,Bruno,1,2200,21
89,Márcia,1,2300,35
false.
```

Tradução das operações primitivas

O predicado `projeta` pode ser utilizado para implementar projeção:

```
projeta(Nome, MatFunc, Salário):-
   funcionário(MatFunc, Nome, _NumDepto, Salário, _MatGerente).
```

A consulta para obter todas as tuplas da projeção é formulada assim:

```
?- findall( (Nome, MatFunc, Salário),
            projeta(Nome, MatFunc, Salário), Proj ).

Proj = [(Marcelo,13,3000),(Joana,21,2000),(Bruno,35,2200),
        (Wagner,38,1600),(Sílvia,43,2500),(Tiago,61,1700),
        (Márcia,89,2300),(Mirela,42,1700)].
```

Composição de operações primitivas

A composição das duas operações anteriores, seleção e projeção, pode ser expressa assim:

```
sel_depois_proj(Nome, MatFunc, Salário):-
    funcionário(MatFunc, Nome, 1, Salário, _MatGerente),
    Salário > 2100.
```

Ou podemos entrar com isso diretamente como uma consulta:

```
?- funcionário( F, N, 1, S, _), S > 2100, write( (N, F, S) ), nl, fail.

Bruno,35,2200
Márcia,89,2300
false.
```

Junção de relações em Prolog

Podemos realizar a *junção* de *Funcionário* com *Departamento* que possum o mesmo número de departamento, assim:

```
junta(MatFunc, NomeF, NumDeptoF, Salário, MatGerenteF,
      NumDeptoF, NomeDepD, MatGerenteD):-
    funcionário(MatFunc, NomeF, NumDeptoF, Salário, MatGerenteF),
    departamento(NumDeptoF, NomeDepD, MatGerenteD).
```

Uma possível consulta seria:

```
?- junta(MatFunc, NomeF, NumDeptoF, Salário, MatGerenteF,
         NumDeptoF, NomeDepD, MatGerenteD),
   write( (MatFunc, NomeF, NumDeptoF, Salário, MatGerenteF,
           NumDeptoF, NomeDepD, MatGerenteD)), nl, fail.

21,Joana,1,2000,13,1,Relações Públicas,21
35,Bruno,1,2200,21,1,Relações Públicas,21
38,Wagner,1,1600,35,1,Relações Públicas,21
43,Sílvia,2,2500,13,2,Segurança,43
61,Tiago,1,1700,21,1,Relações Públicas,21
...
```

Operações sobre conjuntos

Sejam `a(X1, ..., Xn)` e `b(X1, ..., Xn)` predicados que atuam como geradores de tuplas, tais como `departamento(D,N,G)` ou `projeta(N,F,S)`. Temos então

```
a_união_b(X1, ..., Xn):-
    a(X1, ..., Xn) ; b(X1, ..., Xn).

a_interseção_b(X1, ..., Xn):-
    a(X1, ..., Xn) , b(X1, ..., Xn).

a_diferença_b(X1, ..., Xn):-
    a(X1, ..., Xn) , \+ b(X1, ..., Xn).
```

Funções de agregação

Consultas envolvendo apenas operações primitivas podem ser respondidas sem de fato criar a relação resultante.

- Suas tuplas podem ser geradas por um laço dirigido por falha e mostrados imediatamente.
- Ou podem ser geradas diretamente, uma a uma, usando *backtracking*.

Entretanto, para o cálculo de uma função de agregação precisamos de todo o atributo (uma coluna) de uma só vez. Podemos resolver isso usando `bagof`, `findall` ou `setof`. Por exemplo:

```
?- bagof(Salário, funcionário(_, _, _, Salário, _), Salários),
   max_list(Salários, MaiorSalário), write(MaiorSalário), nl.

3000
Salários = [3000],
MaiorSalário = 3000 .
```

Otimização de consultas

Otimização de consultas

Ao se utilizar um banco de dados podemos realizar consultas que, embora corretas, possuam eficiências computacionais diferentes.

Example

Um crime foi cometido e um homem em um Ford azul está sendo procurado. O banco de dados da polícia possui duas tabelas: uma com 3000 carros e outra com 10 000 pessoas suspeitas. Lembre-se que uma pessoa pode possuir mais de um carro.

Imagine que haja 10 fords azuis e que a metade das pessoas sejam homens. Há duas formas de formular uma consulta:

```
findall( (Nome, Placa),
         ( carro(Placa, ford, Nome, azul), pessoa(Nome, m) ),
         Suspeitos).
```

```
findall( (Nome, Placa),
         ( pessoa(Nome, m), carro(Placa, ford, Nome, azul) ),
         Suspeitos).
```

Otimização de consultas

No primeiro caso

```
findall( (Nome, Placa),
         ( carro(Placa, ford, Nome, azul), pessoa(Nome, m) ),
         Suspeitos).
```

serão realizadas 3000 tentativas de unificação de carros, das quais apenas 10 serão bem sucedidas, pois só há 10 fords azuis, produzindo 10 acessos à tabela de pessoas para verificar o sexo, num total de $3000 + 10 = 3010$ unificações.

No segundo caso

```
findall( (Nome, Placa),
         ( pessoa(Nome, m), carro(Placa, ford, Nome, azul) ),
         Suspeitos).
```

serão realizadas 10 000 tentativas de unificação de pessoas, das quais 5000 serão bem sucedidas. Para cada unificação bem sucedida, 3000 outros acessos deverão ser feitos aos carros. O total de unificações será $5000 \times 3000 + 10 = 15\,000\,010$.

Otimização de consultas

Do exemplo mostrado podemos concluir que os predicados que produzam um menor número de soluções possíveis deveriam ser colocados em primeiro lugar nas consultas realizadas.

Otimização de consultas

Algumas vezes teremos que usar `bagof` por questões de eficiência.

Considere o seguinte exemplo: queremos encontrar todos os funcionários do departamento 1 que ganhem no máximo 1000 e que sejam sócios do clube da empresa desde 2010.

```
?- funcionário(F, N, 1, S, _), S =< 1000,
   clube(F, _, _, DataIngresso), DataIngresso >= 2010,
   write( (F, N) ), nl, fail.
```

O predicado `clube` pode ser acessado muitos vezes, o que pode acarretar uma demora. Para resolver isso, podemos pré calcular o conjunto de sócios do clube:

```
?- bagof(Funcionário,
         ( clube(Funcionário, _, _, DataIngresso), DataIngresso >= 2010 ),
         Sócios),
   funcionário(F, N, 1, S, _), S =< 1000,
   member(F, Sócios),
   write( (F, N) ), nl, fail.
```

Consultas não relacionais

Consultas não relacionais

Existem consultas que não podem ser expressas como uma composição de seleções, projeções, junções e funções de agregação.

Example

Encontrar todos os funcionários que ganhem pelo menos tão bem quanto algum de seus superiores.

A relação é um superior de é transitiva, mas podemos expressar apenas as relações é um gerente imediato de, é um gerente imediato de um gerente imediato de, etc.

Este tipo de consulta não oferece dificuldade para o Prolog. Podemos definir um predicado para gerar os salários dos gerentes:

```
salário_gerente(MatGerente, Salário):-
    funcionário(MatGerente, _, _, Salário, _).
salário_gerente(MatGerente, Salário):-
    funcionário(MatGerente, _, _, _, MatGerenteDoGerente),
    salário_gerente(MatGerenteDoGerente, Salário).
```

Visões

Visões

Uma relação que é calculada ao invés de ser simplesmente armazenada é chamada de visão ou vista.

Uma visão é a resultante das operações primitivas aplicadas diretamente nas relações armazenadas ou indiretamente aplicadas a outras visões. Dito de outra forma, elas são *tabelas virtuais*.

Uma visão é alterada sempre que as relações sobre as quais é baseada sofrem alguma modificação.

A relação `salário_gerente`, por exemplo, somente pode ser obtida pelo encaixamento de operações primitivas em uma linguagem de programação com recursão ou iteração.

Uma vantagem do Prolog é sua habilidade de expressar tuplas, visões, consultas e programas especiais na mesma linguagem.

Visões

Essa vantagem oferece a possibilidade de reforçar restrições de integridade que são condições específicas de uma aplicação para garantir a exatidão e a consistência dos dados.

As restrições devem ser testadas antes que se realize qualquer alteração na relação. Por exemplo, podemos usar o predicado a seguir para inserir somente tuplas corretas:

```
insere(Tupla):-
   verifica_inserção(Tupla), !, assertz(Tupla).
insere(Tupla):- notifica_violação(Tupla).

verifica_inserção(funcionário(F, _, D, _, G)):-
   !,
   F =\= G, % um funcionário não pode ser gerente dele mesmo
   departamento(D, _, _). % o departamento existe
verifica_inserção(_). % as outras inserções estão OK
```

Persistência de dados

Persistência de dados

Discutimos até agora formas de se construir em Prolog um banco de dados em memória durante uma execução. Precisamos aumentar nosso modelo para endereçar os seguintes problemas:

1. Os dados precisam sobreviver entre execuções do programa, ou seja, precisamos que os dados sejam persistentes.
2. Em várias aplicações, tal como um serviço web, pode haver acessos simultâneos à mesma informação.
    - Isto normalmente não será um problema se o acesso for somente para leitura.
    - Entretanto, para a **escrita** a situação pode rapidamente se tornar complexa e o resultado final dependerá de quem conseguiu acessar por último.

O primeiro problema pode ser resolvido por meio dos predicados de manipulação de arquivos já estudados.

O segundo problema, chamado de acesso concorrente aos dados, pode ser resolvido usando mecanismos de controle de concorrência. Aqui, usaremos mutex.

A biblioteca `persistency`

O SWI-Prolog nos oferece a biblioteca `persistency` que permite o armazenamento persistente simples de um ou mais predicados dinâmicos.

Com ela pode-se adicionar ou remover fatos de um arquivo usado como um armazém de fatos persistentes.

Para a `persistency` Uma tabela sempre está associada a um módulo.

Um módulo que mantenha uma tabela deve declarar o esquema para a relação que será armazenada usando a diretiva `persistent/1`.

Exemplo de uso de `persistency`

Vamos retomar a relação *Funcionário* como exemplo. Colocaremos todas as definições no arquivo `funcionário.pl`:

```
:- module(
       funcionário, % nome do módulo
       [ % aqui são colocados os predicados a serem exportados
         % no formato funtor/aridade.
       ]).

% importa a biblioteca persistency
:- use_module(library(persistency)).

% esquema da relação
:- persistent
   funcionário(matrícula:positive_integer,
               nome:atom,
               departamento:nonneg,
               salário:positive_integer,
               matrícula_gerente:positive_integer).

% Anexa o arquivo que servirá como armazém de fatos
:- initialization(db_attach('tbl_funcionário.pl', [])).
```

Exemplo de uso de `persistency`

```
:- persistent
   funcionário(matrícula:positive_integer,
               nome:atom,
               departamento:nonneg,
               salário:positive_integer,
               matrícula_gerente:positive_integer).
```

A diretiva `persistent//1` expande cada declaração em quatro predicados:

funcionário//5 um fato dinâmico com o mesmo funtor e aridade da declaração.

assert_funcionário//5 este predicado permite a inserção de novas tuplas (fatos) na relação.

retract_funcionário//5 para remover um fato que unificar com os argumentos.

retractall_funcionário//5 para remover todos os fatos que unificarem com os argumentos.

Controle de concorrência usando mutex

Para organizar o acesso simultâneo aos dados existem os mecanismos de controle de concorrência que:

- Limitam o acesso concorrente aos dados e recursos compartilhados.
- Buscam evitar inconsistências nos dados causadas pelo acesso concorrente.

Dentre os vários mecanismos de controle de concorrência existentes há o mutex, também chamado de lock.

Mutex é o acrônimo *mutual exclusion* (exclusão mútua) que evita que dois ou mais processos ou *threads* tenham acesso simultâneo a um recurso compartilhado.

A exclusão mútua é garantida via a criação de uma fila de acesso ao recurso compartilhado:

- Permite somente um acesso por vez
- Caso o dado ou recurso esteja em uso, o processo que tentar acessá-lo será colocado em um fila até que o recurso seja liberado.

Mutex em SWI-Prolog

Em SWI-Prolog o mutex é implementado pelo predicado `with_mutex(Chave,Predicado)`:

- O primeiro processo a entrar, retém a *Chave*, um átomo, e ganha a permissão para executar o *Predicado*.
- Os demais aguardarão em uma fila até que a chave seja liberada.
- A chave é liberada após a execução do *Predicado* terminar.
- Após a chave ser liberada algum processo da fila ganha acesso à *Chave* e pode executar seu *Predicado*.
- *Predicado* pode ser uma sequência de chamadas a outros predicados: `( pred1, pred2, ..., predN ).`

Uso de with_mutex e persistency

Voltando ao exemplo da relação *Funcionário*, podemos agora implementar a operação de inserção de uma nova tupla:

```
insere(MatFunc, Nome, Departamento, Salário, MatrículaGerente):-
    with_mutex(funcionário,
               assert_funcionário(MatFunc, Nome, Departamento,
                                  Salário, MatrículaGerente)).
```

Qualquer processo que executar `insere` reterá a chave `funcionário` e poderá inserir um novo fato tanto na memória quanto no arquivo.

Para remover um fato usaremos `remove`. Note que a chave `funcionário` é a mesma de `insere` e isto significa que estas operações não poderão ser executadas simultaneamente.

```
remove(MatFunc):-
    with_mutex(funcionário,
               retract_funcionário(MatFunc, _Nome, _Departamento,
                                   _Salário, _MatrículaGerente)).
```

Uso de with_mutex e persistency

Podemos implementar a operação de atualização assim:

```
atualiza(MatFunc, Nome, Departamento, Salário, MatrículaGerente):-
    with_mutex(funcionário,
                ( retractall_funcionário(MatFunc, _Nome, _Departamento,
                                         _Salário, _MatrículaGerente),
                  assert_funcionário(MatFunc, Nome, Departamento,
                                     Salário, MatrículaGerente)) ).
```

Novamente, reutilizamos a chave `funcionário`. Além disso, a atualização é feita em dois passos: remove a tupla antiga e depois adiciona a nova.

Assumindo que o código para a manipulação da relação *Funcionário* está no arquivo `funcionário.pl`, usaremos a implementação entrando no intérprete do Prolog e consultando este arquivo.

```
?- [funcionário].
true.
```

Exemplo de uso da implementação de Funcionário

Podemos verficar que os predicados dinâmicos foram criados:

```
?- listing(funcionário).
:- dynamic funcionário:funcionário/5.
```

O formato aqui é módulo:funtor/aridade. O módulo é `funcionário`, o funtor é `funcionário` com aridade 5.

Como ainda não inserimos tupla alguma, uma consulta de leitura resultará `false`:

```
?- funcionário:funcionário(MatFunc, Nome, Depto, Salário, MatGerente).
false.
```

Vamos agora adicionar a primeira tupla:

```
?- funcionário:insere(13, 'Marcelo', 0, 3000, 19).
```

Se repetirmos a penúltima consulta, obteremos

```
?- funcionário:funcionário(MatFunc, Nome, Depto, Salário, MatGerente).
MatFunc = 13, Nome = 'Marcelo', Depto=0, Salário=3000, MatGerente=19.
```

Exemplo de uso da implementação de Funcionário

Vamos agora adicionar mais três tuplas:

```
?- funcionário:insere(21, 'Joana'  , 1, 2000, 13),
   funcionário:insere(35, 'Bruno'  , 1, 2200, 21),
   funcionário:insere(38, 'Wagner' , 1, 1600, 35).
```

Vamos ver o que está na memória agora:

```
listing(funcionário).

funcionário:funcionário(13, 'Marcelo', 0, 3000, 19).
funcionário:funcionário(21, 'Joana', 1, 2000, 13).
funcionário:funcionário(35, 'Bruno', 1, 2200, 21).
funcionário:funcionário(38, 'Wagner', 1, 1600, 35).
```

Vamos remover o funcionário cuja matrícula é 35:

```
?- funcionário:remove(35).

?- findall((F, N, D, S, G), funcionário:funcionário(F, N, D, S, G), Func).
Func = [(13, 'Marcelo', 0, 3000, 19),  (21, 'Joana', 1, 2000, 13),
        (38, 'Wagner', 1, 1600, 35)].
```

Exemplo de uso da implementação de Funcionário

Vamos olhar como está o arquivo `tbl_funcionário.pl` que é o arquivo associado à tabela *Funcionário*:

```
created(1600174128.4267464).
assert(funcionário(13,'Marcelo',0,3000,19)).
assert(funcionário(21,'Joana',1,2000,13)).
assert(funcionário(35,'Bruno',1,2200,21)).
assert(funcionário(38,'Wagner',1,1600,35)).
retract(funcionário(35,'Bruno',1,2200,21)).
```

Aqui vemos um fato, `created/1`, que é criado pelo Prolog para indicar a data de criação ou modificação do arquivo em segundos deste 1/1/1970.

Note que neste arquivo estão os predicados necessários para reestabelecer a base de dados na memória.

Exemplo de uso da implementação de Funcionário

Em particular, note que ao invés de realmente apagar o funcionário 35, ele deixou o predicado responsável por isto:

```
retract(funcionário(35,'Bruno',1,2200,21)).
```

Com o passar do tempo este arquivo pode ficar muito grande, cheio de retracts. Há uma forma de limpar o arquivo, bastando acrescentar em `funcionário.pl` o seguinte:

```
sincroniza :-
    db_sync(gc(always)).
```

Se sairmos do Prolog e depois voltarmos, podemos então digitar

```
?- [funcionário].

?- funcionário:sincroniza.
```

Exemplo de uso da implementação de Funcionário

Depois de sincronizarmos, o arquivo `tbl_funcionário.pl` agora estará assim:

```
created(1600178782.562746).
assert(funcionário(13,'Marcelo',0,3000,19)).
assert(funcionário(21,'Joana',1,2000,13)).
assert(funcionário(38,'Wagner',1,1600,35)).
```

Note que não há mais `retract` algum no arquivo.

Modelagem de dados

Modelagem de dados

Um banco de dados não se resume apenas a uma coleção de dados ou entidades, mas contém também associações ou relacionamentos entre estas entidades.

Um modelo entidade relacionamento é um modelo de dados relacional que permite a descrição sistemática de um processo. O modelo relacional baseia-se em três conceitos:

entidade objeto existente no mundo real com uma identificação distinta e com um significado próprio.

atributo é uma característica ou qualidade de uma entidade

relacionamento é uma associação entre as entidades determinada pelo modo na qual cada tupla em uma relação está ligada a tuplas em outra relação.

No modelo relacional há a criação de um modelo lógico consistente da informação a ser armazenada. Este modelo lógico pode ser simplificado por meio de um processo chamado de normalização.

Normalização

A normalização consiste numa série de passos que visa a organização de um projeto de banco de dados para, entre outras razões:

- Permitir um método simples e padronizado para lidar com a complexidade conceitual.
- Reduzir a redundância de dados verificando onde os mesmos dados são repetidos em diferentes lugares. Dados duplicados aumentam fortemente o risco de erro quando fazemos atualizações no banco de dados.
- Tornar a atualização, remoção e inserção de dados mais fáceis.
- Facilitar a compreensão da estrutura lógica do banco de dados para usuários fora da área computacional.

Os passos para a simplificação de um banco de dados são chamados de formas normais. As diferentes formas normais são chamadas de: *primeira forma normal*, *segunda forma normal*, etc. Aqui abordaremos as três primeiras.

Forma original dos dados

Considere que a coleção de dados a seguir foi encontrada para um determinado domínio de problema.

Id depto	Nome depto	Localização	Id máquina	Nome	Tipo	Responsável	Ramal
P1	Produção	Fábrica1	P100	Torno	Confecção	Horácio	456
			P101	Broca			
			P102	Serra			
			P103	Desktop	Computador	Luís	789
V2	Vendas	Anexo	V121	Copiadora	Escritório	Maria	213
			V125	Impressora			
F1	Financeiro	Central	F234	Desktop	Computador	Luís	789
			P123	Copiadora	Escritório	Maria	213
			F101	Impressora			

Cada trecho em branco significa que é o mesmo item visto por último na coluna.

Primeira forma normal (1FN)

Os passos necessários para garantir que os dados estejam na primeira forma normal são:

1. Assegurar que todos os argumentos estão expressos como elementos simples de dados.
2. Assegurar que há somente um valor para cada argumento, ou seja, não há grupos ou sublistas.
3. Explicitar todos os valores de dados, completando os dados onde o valor é subentendido.

Aplicando a 1FN aos dados originais do slide anterior, obteremos a tabela mostrada no próximo slide.

Primeira forma normal (1FN)

Id depto	Nome depto	Localização	Id máquina	Nome	Tipo	Responsável	Ramal
P1	Produção	Fábrica1	P100	Torno	Confecção	Horácio	456
P1	Produção	Fábrica1	P101	Broca	Confecção	Horácio	456
P1	Produção	Fábrica1	P102	Serra	Confecção	Horácio	456
P1	Produção	Fábrica1	P103	Desktop	Computador	Luís	789
V2	Vendas	Anexo	V121	Copiadora	Escritório	Maria	213
V2	Vendas	Anexo	V125	Impressora	Escritório	Maria	213
F1	Financeiro	Central	F234	Desktop	Computador	Luís	789
F1	Financeiro	Central	P123	Copiadora	Escritório	Maria	213
F1	Financeiro	Central	F101	Impressora	Escritório	Maria	213

Embora os dados ainda estejam na 1FN já é possível representá-los em Prolog com

```
maquinário(DRef, NomeDep, Localização, IdMáquina, Nome, Tipo, Resp, Ramal).
```

As duas primeiras linhas ficariam assim:

```
maquinário(p1, produção, fábrica1, p100, torno, confecção, horácio, 456).
maquinário(p1, produção, fábrica1, p101, broca, confecção, horácio, 456).
```

Dependência funcional

Um dado é funcionalmente dependente se ele é diretamente dependente de um outro dado. Dessa forma, uma dependência funcional $X \longrightarrow Y$ significa que os valores de Y são determinados pelo valores de X ou, equivalentemente Y é funcionalmente dependente de X.

A ideia da dependência funcional é emprestada da matemática:

- $y = f(x) = 5x + 2$ y aqui é funcionalmente dependente de x, pois existe uma função mapeando valores de x em y. Se você sabe o valor x, então também sabe o valor de y.
- Há uma dependência funcional entre o *CPF* é o *nome* de uma pessoa. Ou seja, o *nome* é função do *CPF*: $f(\text{CPF}) = \text{nome}$ ou $f : \text{CPF} \longrightarrow \text{nome}$

Regras para encontrar dependências funcionais

Separação se $A \longrightarrow B, C$ então $A \longrightarrow B$ e $A \longrightarrow C$.

Exemplo:

se CPF $\longrightarrow$ nome, endereço então CPF $\longrightarrow$ nome e CPF $\longrightarrow$ endereço.

Isto pode ser lido assim: *se com um número de CPF eu encontro o nome e o endereço de uma pessoa, então com este mesmo número eu posso encontrar apenas o nome e apenas o endereço.*

Acumulação se $A \longrightarrow B, C$ então $A, C \longrightarrow B$.

Exemplo:

se CPF $\longrightarrow$ endereço então CPF, idade $\longrightarrow$ endereço. Isto pode ser lido assim: *se com um número de CPF eu encontro o endereço de uma pessoa, então com este mesmo número acrescido da idade da pessoa eu posso encontrar o endereço também.*

Regras para encontrar dependências funcionais

Transitividade se $A \longrightarrow B$ e $B \longrightarrow C$ então $A \longrightarrow C$.

Exemplo:

se CPF $\longrightarrow$ CEP e CEP $\longrightarrow$ cidade então CPF $\longrightarrow$ cidade. Isto pode ser lido assim: *se com um número de CPF eu encontro o CEP de uma cidade e com o CEP eu encontro a cidade, então com o CPF eu encontro a cidade.*

Pseudo-transitividade se $A \longrightarrow B$ e $B, C \longrightarrow D$ então $A, C \longrightarrow D$.

Exemplo:

se CPF $\longrightarrow$ código-funcionário e código-funcionário, mês $\longrightarrow$ salário então CPF, mês $\longrightarrow$ salário.

Lemos assim: *se com um CPF eu encontro o código do funcionário e com o código do funcionário acrescido de um certo mês eu encontro o salário recebido por ele neste mês, então com o CPF acrescido por um certo mês eu encontro o salário recebido por ele neste mês.*

Segunda forma normal (2FN)

Uma relação está na segunda forma normal se, e somente se:

1. ela estiver na 1FN.
2. cada atributo não-chave for funcionalmente dependente da chave primária inteira, ou seja, cada atributo não-chave não poderá ser dependente de apenas parte da chave.

Nas relações com chave primária composta, se um atributo depende apenas de uma parte da chave primária, então esse atributo deve ser colocado em outra relação.

Segunda forma normal (2FN)

Retomemos os dados de parte da relação *Maquinário* na 1FN:

Id depto	Nome depto	Localização	Id máquina	Nome	Tipo	Responsável	Ramal
P1	Produção	Fábrica1	P100	Torno	Confecção	Horácio	456
V2	Vendas	Anexo	V125	Impressora	Escritório	Maria	213
F1	Financeiro	Central	F234	Desktop	Computador	Luís	789

A relação *Maquinário* possui uma chave composta pelos atributos *Identificador do Departamento* e *Identificador da Máquina*. Assim, para colocar esta relação na 2FN teremos que particioná-la.

O *Nome do Departamento* e a *Localização* são funcionalmente dependentes do *Identificador do Departamento* e, assim, podemos agrupar esses três atributos em uma nova relação chamada *Departamento*. Após o agrupamento, removemos todas as tuplas duplicadas. A nova relação é mostrada no próximo slide.

Segunda forma normal (2FN)

Departamento

Id depto	Nome depto	Localização
P1	Produção	Fábrica1
V2	Vendas	Anexo
F1	Financeiro	Central

Os atributos *Nome*, *Tipo*, *Responsável* e *Ramal* são funcionalmente dependentes da chave *Identificador da Máquina* e, portanto, criaremos uma nova relação:

Máquina

Id máquina	Nome	Tipo	Responsável	Ramal
P100	Torno	Confecção	Horácio	456
P101	Broca	Confecção	Horácio	456
P102	Serra	Confecção	Horácio	456
P103	Desktop	Computador	Luís	789
V121	Copiadora	Escritório	Maria	213
V125	Impressora	Escritório	Maria	213
F234	Desktop	Computador	Luís	789
P123	Copiadora	Escritório	Maria	213
F101	Impressora	Escritório	Maria	213

Segunda forma normal (2FN)

Para restabelecer o vínculo existente entre a chave composta por *Identificador do Departamento* e *Identificador da Máquina*, criaremos uma terceira relação chamada Maquinário Departamento. Isto cria um relacionamento entre máquinas e os departamentos onde elas se encontram.

Maquinário Departamento

Id depto	Id máquina
P1	P100
P1	P101
P1	P102
P1	P103
V2	V121
V2	V125
F1	F234
F1	P123
F1	P101

Terceira forma normal (3FN)

Uma relação R está na terceira forma normal se:

1. ela estiver na 2FN.
2. cada atributo não-chave de R deve possuir independência transitiva relativo a qualquer outro atributo não-chave de R. Ou seja, todos os atributos não-chaves de R devem ser independentes uns dos outros e funcionalmente dependentes apenas da chave primária de R.

Retomando parte da relação *Máquina*:

Id máquina	Nome	Tipo	Responsável	Ramal
P100	Torno	Confecção	Horácio	456
F101	Impressora	Escritório	Maria	213

Note que os atributos *Ramal* e *Responsável* são funcionalmente dependentes do atributo não-chave *Tipo* e, assim, devem ser colocados em uma relação separada que chamaremos de *Responsabilidade*. Novamente, todas as duplicações devem ser removidas.

Terceira forma normal (3FN)

Responsabilidade

Tipo	Responsável	Ramal
Confecção	Horácio	456
Computador	Luís	789
Escritório	Maria	213

Com esta separação, o que restou da relação *Máquina* ficará em uma nova relação chamada *Máquina Tipo*.

Máquina Tipo

Id máquina	Nome	Tipo
P100	Torno	Confecção
P101	Broca	Confecção
P102	Serra	Confecção
P103	Desktop	Computador
V121	Copiadora	Escritório
V125	Impressora	Escritório
F234	Desktop	Computador
P123	Copiadora	Escritório
F101	Impressora	Escritório

Formas normais

As quatro relações resultantes da 3FN a saber, *Departamento*, *Maquinário Departamento*, *Responsabilidade* e *Máquina Tipo*, podem ser facilmente implementadas em Prolog usando as técnicas vistas nesta aula.

Além disso, se uma nova máquina precisar ser adicionada ao banco de dados os únicos detalhes que devem ser conhecidos são aqueles da relação *Máquina tipo*:

Id máquina	Nome	Tipo
P274	Notebook	Escritório

Se usássemos os dados na 1FN, então também teríamos que saber: *Id depto*. *Nome depto*, *Localização*, *Responsável* e *Ramal*.

Isso, por sua vez, envolve decidir para qual departamento a máquina vai. Se não soubermos, a máquina não pode ser registrada. Isto também envolve duplicar a *Responsabilidade* que poderia levar à perda de consistência de dados no caso de alguma entrada incorreta ser feita.

Para saber mais

Para saber mais

- Banco de dados relacional. In: *Wikipédia, a enciclopédia livre*. Flórida: Wikimedia Foundation, 2020. Disponível em: pt.wikipedia.org/w/index.php?title=Banco_de_dados_relacional&oldid=59076050.

- Visão (banco de dados). In: *Wikipédia, a enciclopédia livre*. Flórida: Wikimedia Foundation, 2018. Disponível em: pt.wikipedia.org/w/index.php?title=Vis%C3%A3o_(banco_de_dados)&oldid=52667010.

- Álgebra Relacional. In: *Plataforma DevMedia, 2020*. Disponível em www.devmedia.com.br/algebra-relacional-parte-i/2663 e www.devmedia.com.br/linguagem-de-consulta-formal-algebra-relacional-parte-ii/20123

- Chave primária, chave estrangeira e candidata. Disponível em https://www.luis.blog.br/chave%20primaria-chave-estrangeira-e-candidata.html

Para saber mais

- Modelo relacional. In: *Wikipédia, a enciclopédia livre*. Flórida: Wikimedia Foundation, 2020. Disponível em: pt.wikipedia.org/w/index.php?title=Modelo_relacional&oldid=57833728.

- Normalização de dados. In: *Wikipédia, a enciclopédia livre*. Flórida: Wikimedia Foundation, 2020. Disponível em: pt.wikipedia.org/w/index.php?title=Normaliza%C3%A7%C3%A3o_de_dados&oldid=58577823.

- André Rodrigo Sanches. *Modelo físico*. Disponível em https://www.ime.usp.br/~andrers/aulas/bd2005-1/aula11.html

Referências bibliográficas

Referências bibliográficas

- Goble, Terry. *Structured systems analysis through Prolog*. 1989. Prentice Hall.
- Kluźniak, Feliks e Szpakowicz, Sanislaw. *Prolog for programmers*. 1985. Academic Press.
- Palazzo, Luiz A. M. *Introdução à programação Prolog*. 1997. Educat.