Capítulo 8: BDs Objecto-Relacional

- Tipos de dados complexos e objectos
- Tipos de dados estruturados e herança em SQL
- Herança de tabelas
- Matrizes e multi-conjuntos em SQL
- Identidade de Objectos e Referência em SQL
- Implementação das características O-R
- Linguagens de Programação Persistentes
- Comparação entre os vários modelos

Modelo de Dados Objecto-Relacional

- Tem como base o modelo relacional, e generaliza-o introduzindo noções de objectos e formas de lidar com tipos complexos.
- Permite que os atributos tenham valores de tipos complexos, incluindo valores não atómicos.
- Aumenta a capacidade de modelação, sem perder as principais vantagens do modelo relacional (incluindo o acesso declarativo aos dados).
- É compatível com as linguagens relacionais existentes (o que torna este modelo bastante apelativo para sistemas de bases de dados comerciais).

Relações imbricadas

- Motivação:
  - Permitir domínios não atómicos (atómico ≡ indivisível)
  - Exemplo de domínios não atómicos:
    - Conjunto de inteiros
    - Conjunto de tuplos
  - Permite uma modelação mais intuitiva em aplicações com tipos complexos
- Definição informal:
  - Permitir relações em todo o local onde antes se permitiam valores atómicos — relações dentro de relações
  - Manter base matemática do modelo relacional
  - Viola a 1ª forma normal

Exemplo de relação imbricada

- Numa biblioteca cada livro tem:
  - um título,
  - um *conjunto* de autores,
  - uma editora,
  - um *conjunto* de keywords
- Uma instância da relação *livros,* que *não está* na 1ª Forma Normal, é:

title	*author-set*	*publisher*	*keyword-set*
		(*name, branch*)	
Compilers	{Smith, Jones}	(McGraw-Hill, New York)	{parsing, analysis}
Networks	{Jones, Frick}	(Oxford, London)	{Internet, Web}

Versão 1NF de relação imbricada

- Na 1NF a instância da relação *livros* seria:

title	*author*	*pub-name*	*pub-branch*	*keyword*
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web

livros-flat

Decomposição 4NF de Relações Imbricadas

- Remover redundâncias de *livros-flat* assumindo as seguintes dependências multivalor:
  - *título* $\twoheadrightarrow$ *autor*
  - *título* $\twoheadrightarrow$ *keyword*
  - *título* $\twoheadrightarrow$ *nome-pub, pub-branch*
- Com decomposição de *livros-flat* para a 4NF obtêm-se os esquemas:
  - *(título, autor)*
  - *(título, keyword)*
  - *(título, nome-pub, pub-branch)*

Decomposição *livros-flat* para a 4NF

title	*author*
Compilers	Smith
Compilers	Jones
Networks	Jones
Networks	Frick

authors

title	*keyword*
Compilers	parsing
Compilers	analysis
Networks	Internet
Networks	Web

keywords

title	*pub-name*	*pub-branch*
Compilers	McGraw-Hill	New York
Networks	Oxford	London

books4

Problemas com a 4NF

- Os esquemas na 4NF precisam de junção para grande parte das perguntas.
- A relação na 1NF, *livros-flat,* que corresponde à junção das relações na 4NF:
  - Elimina a necessidade de junções,
  - Mas perde a correspondência biunívoca entre tuplos e entidades (livros).
  - E tem imensas redundâncias
- Neste caso, a relação imbricada parece ser mais natural.

Tipos complexos no SQL:1999

- As extensões do SQL para suportar tipos complexos incluem:
  - Colecções e objectos de grande dimensão
    - Relações imbricadas são um exemplo de colecções
  - Tipos estruturados
    - Tipos com atributos compostos (registos)
  - Herança
  - Orientação por objectos
    - Incluindo identificadores de objectos e referências
- A apresentação que se segue baseia-se no standard SQL:1999 (com algumas extensões)
  - Não está (ainda) completamente implementado em nenhum sistema comercial
  - Mas algumas das características existem já na maioria dos sistemas
    - O Oracle já suporta boa parte do que se vai apresentar

Tipos Estruturados e Herança em SQL

- Tipos Estruturados podem ser declarados e usados em SQL

```
create type NameType as
    (firstname      varchar(20),
     lastname       varchar(20))
     final

create type AddressType as
    (street         varchar(20),
     city           varchar(20),
     zipcode        varchar(20))
     not final
```

  - Nota: **final** e **not final** indica se se podem criar subtipos

- Tipos estruturados podem ser usados para criar tabelas com atributos compostos

```
create table customer (
    name        NameType,
    address     AddressType,
    dateOfBirth date)
```

- Usa-se a notação com "." para referir os componentes: *name.firstname*

Tipos Estruturados (cont.)

- Tipos de linha (row-type) definidos pelo utilizador

  create type *CustomerType* **as** (
  name NameType,
  address AddressType,
  dateOfBirth **date**)
  not final

- Podemos então criar uma tabela cujas linhas são do tipo definido pelo utilizador

  create table *customer* **of** *CustomerType*

- Em alternativa, podemos usar tipos anónimos

  create table *customer_r(*
  name **row**(first*name* **varchar**(20),
  lastname **varchar**(20)),
  address ***row****(street* **varchar**(20),
  city **varchar**(20),
  zipcode **varchar**(20)),
  dateOfBirth **date**)

Métodos

- Podemos adicionar a declaração de métodos com um tipo estruturado.

  method *ageOnDate* (*onDate* **date**)

  returns interval year

- O corpo de método é dado em separado.

  create instance method *ageOnDate* (*onDate* **date**)

  returns interval year

  for *CustomerType*

  begin

  return *onDate* - **self**.*dateOfBirth*;

  end

- Podemos agora encontrar a idade de cada cliente:

  select *name.lastname, ageOnDate* (**current_date**)

  from *customer*

Funções Constructoras

- **Funções contructoras** são usadas para criar valores de tipos estruturados
- E.g.
  create function *NameType*(*firstname* **varchar**(20), *lastname* **varchar**(20))
  returns *NameType*
  begin
  set self.*firstname = firstname;*
  set self.*lastname = lastname;*
  end
- Para criar um valor do tipo *NameType,* usa-se
  new *NameType*('John' , 'Smith')
- Normalmente usado em inserções
  insert into *customer* **values**
  (**new** *NameType*('John' , 'Smith),
  new *AddressType*(' 20 Main St' , 'New York' , '11001'),
  '1960-8-22');

Herança

- Considere a seguinte definição de tipo para pessoas:

```
create type Pessoa
    (nome varchar(20),
     morada varchar(20))
not final;
```

- Pode-se usar herança para definir os tipos para *estudante* e *docente*

```
create type Estudante under Pessoa
 (grau          varchar(20),
  departamento  varchar(20))
not final;

create type Docente under Pessoa
 (salário          integer,
  departamento     varchar(20))
not final;
```

- Os subtipos podem redefinir os métodos usando (explicitamente) na definição do tipo **overriding method** em vez de **method**

Herança Múltipla

- Num sistema com herança múltipla, pode-se definir monitor como:

  create type *Monitor*
  under *Estudante, Docente*
  final;

- Para evitar conflito entre duas ocorrências de departamento (departamento onde estuda vs departamento onde lecciona), podem-se renomear os atributos

  create type *Monitor*
  under
  Estudante **with** (*departamento* **as** *dep-estud*),
  Docente **with** (*departamento* **as** *dep-docente*)
  final;

- Nem o SQL:1999, nem o SQL:2003, nem o Oracle, suportam herança múltipla
  - Mas há generalizações do SQL que consideram herança múltipla

Herança em Tabelas

- As tabelas criadas de subtipos podem ser especificadas como sub-tabelas.
- *E.g.* **create table** *pessoas* **of** *Pessoa*

  create table *estudantes* **of** *Estudante* **under** *pessoas*
  create table *docentes* **of** *Docente* **under** *pessoas*
- Cada tuplo numa subtabela pertence (implicitamente) à supertabela correspondente i.e. são visíveis a consultas na supertabela.
- *Conceptualmente, a herança múltipla é possível com tabelas*
  - *Monitores como subtabela de estudantes e docentes*
  - *Mas não é suportada pelo SQL*
    - *Não é possível criar um tuplo em pessoas que seja simultaneamente estudante e docente.*

Requisitos de consistência nas subtabelas

- Requisitos de consistência para subtabelas

1. Cada tuplo da supertabela (e.g. pessoas) só pode corresponder, no máximo, a um tuplo em cada uma das subtabelas (e.g. estudantes e docentes)
   - Caso contrário, seria possível ter dois tuplos em estudantes correspondendo à mesma pessoa
2. Todos os tuplos com o mesmo valor nos atributos herdados têm que ser derivados de um só tuplo (inserido numa só tabela).
   - Requisito adicional no SQL:1999
   - Ou seja, cada entidade tem que ter um tipo mais específico
   - Não se pode ter um tuplo em pessoas que corresponda simultaneamente a um tuplo de estudantes e a um tuplo de docentes, a não ser que fossem derivados a partir do mesmo tuplo, e.g. inserido na tabela monitor.

- Por outras palavras, em SQL:1999 a herança em tabelas "implementa" especializações disjuntas.

Tipos Array e Multiset em SQL

- Exemplo de uma declaração com colecções de valores Array e Multiset

```
create type Editor as
    (nome           varchar(20),
     branch         varchar(20))
create type Livro as
    (título             varchar(20),
     array-autores      varchar(20) array [10],
     data-publ          date,
     editor             Editor,
     conj-keyword       varchar(20) multiset)
```

  - O Oracle suporta atributos cujo tipo é **table of...**
  - Usando um array para os autores, pode-se guardar a ordem
- Podem-se depois criar uma tabela:

```
create table livros of Livro
```

Criação de colecções de valores

- Construção de Arrays

  array [‘Silberschatz’ ,`Korth’ ,`Sudarshan’]

- Construção de multi-conjuntos

  multiset [‘computers’ ,`sql’ ,`database’]

- Para criar um tuplos da relação *livros*

  (‘Compilers’ , **array**[`Smith’ ,`Jones’],
  new *Editor*(`McGraw-Hill’ ,`New York’),
  multiset[`parsing’ ,`analysis’])

- Para inserir esse tuplo na relação *livros*

  insert into *livros*
  values
  (`Compilers’ , **array**[`Smith’ ,`Jones’],
  new *Editor*(‘McGraw Hill’ ,`New York’),
  multiset[`parsing’ ,`analysis’])

Consultas em tipos estruturados

- Quais os títulos e respectivos nomes da editora de todos os livros.

  select *título*, *editor.nome*
  from *livros*

- Note que *editor* é um atributo (e não uma tabela).
- Tal como para aceder a atributos de uma tabela, também se usa o ponto para aceder a atributos de atributos com tipos estruturados.
- Para aceder a métodos usa-se o ponto e o nome do método seguido dos argumentos entre parêntesis. Os parêntesis devem lá estar mesmo que não haja argumentos.

Perguntas a colecções

- Os atributos com colecções são usados de forma semelhante tabelas, utilizando agora a keyword **unnest**
- Quais os títulos dos livros que têm *database* entre as palavras chave?

  select *título*
  from *livros*
  where 'database' **in** (***unnest****(conj-keyword)*)

- Pode-se aceder a elementos dum array da forma habitual i.e. usando índices. Se soubermos que um dado livro tem 3 autores, podemos escrever:

  select *array-autores*[1], *array-autores*[2], *array-autores*[3]
  from *livros*
  where *título* = `Database System Concepts

- Obter uma relação contendo pares da forma *título,autor* em que *autor* é autor do livro com título *título:*

  select *B.título, A.nome*
  from *livros* **as** *B*, **unnest** (*B.array-autores*) **as** *A(nome)*

- Para reter a ordenação, podemos acrescentar a clausula

  select *B.título, A.nome*
  from *livros* **as** *B*, **unnest** (*B.array-autores*) **with ordinality as** A(nome,posição)

Unnesting

- O processo de transformação de uma relação imbricada numa forma com menos (ou nenhum) atributo do tipo relação é chamado unnesting.
- E.g.

  select *B.título, A.autor, B.editor.nome-pub, B.editor.pub_branch, K.keyword*

  from *livros* **as** *B*, **unnest** (*B.array-autores*) **as** *A(autor),*
  unnest (*B.conj-keyword*) **as** *K(keyword),*

title	*author*	*pub-name*	*pub-branch*	*keyword*
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web

Nesting

- **Nesting** é o processo oposto ao unnesting, criando um atributo do tipo relação.
- O SQL:1999 não suporta nesting.
- O nesting pode ser feito de maneira semelhante às agregações
  - para criar um conjunto, usa-se a função **collect()** em vez da função de agregação, para criar um multi-conjunto.
- Para, em *livros-flat,* agrupar num atributo o conjunto das *keywords*:

  select *título*, *autor*, *Editor(nome-pub, pub_branch*) **as** *editor*,
  collect(*keyword*) **as** *conj-keyword*
  from *livros-flat*
  group by *título, autor, editor*
- Para agrupar também os vários autores dum livro:

  select *título*, **collect**(*autor*) **as** *autores*,
  Editor(nome-pub, pub_branch) **as** *editor*,
  collect(*keyword*) **as** *conj-keyword*
  from *livros-flat*
  group by *título*, *editor*

Versão 1NF de relação imbricada

- Na 1NF a instância da relação *livros* seria:

title	*author*	*pub-name*	*pub-branch*	*keyword*
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web

livros-flat

Nesting (Cont.)

- Outra abordagem consiste em usar sub-perguntas na cláusula **select**:

```
select título,
        array( select M.autor
          from livros-flat as M
          where M.título =O.título) as autores,
        Editor(nome-pub, pub-branch) as editor,
        multiset(select N.keyword
          from livros-flat as N
          where N.título = O.título) as conj-keyword
from livros-flat as O
```

- Pode usar-se **order by** em perguntas *nested* para obter colecções ordenadas, úteis na criação de arrays.

Identidade de Objectos

- Um objecto mantém a sua identidade mesmo que os valores de todas as suas variáveis, e mesmo os seus métodos, mudem.
- A identidade de objectos neste modelo é mais forte que no modelo relacional, ou em linguagens de programação.
- Identidade por:
  - **Valor** – e.g. chaves primárias no modelo relacional.
  - **Nome** – dado pelo utilizador, como nas linguagens de programação.
  - **Built-in** – a identidade está built-in no modelo de dados.
    - O utilizador não precisa fornecer um identificador.
    - É a forma de identidade usada no modelo de objectos.

Identificadores de Objectos

- Os **Identificadores de Objectos** são usados por objectos para identificar univocamente outros objectos
  - São únicos:
    - Não pode haver dois objectos com o mesmo identificador
    - Cada objecto tem apenas um identificador
  - E.g., numa *inscrição* o atributo *num_aluno* é um identificador de um objecto de *alunos* (em vez da inscrição conter o próprio aluno).
  - Podem ser
    - Gerados pela base de dados
    - externos (e.g. número de BI)
  - Os identificadores gerados automaticamente:
    - São mais fáceis de usar
    - Podem ser redundantes se já existe atributo único

Tipos referência

- As linguagens de objectos permitem criar e referenciar objectos.
- No SQL:1999
  - Podem-se referenciar tuplos
  - Cada referência diz respeito a tuplo de uma tabela específica. I.e. cada referência deve limitar o escopo a uma única tabela

Declaração de Referências em SQL:1999

- E.g. definir o tipo *Departamento* com atributo *nome* e atributo *presidente* que é uma referência para o tipo *Pessoa,* tendo a tabela *pessoas* como escopo

  create type *Departamento* **as**(
  nome **varchar(20)**,
  presidente **ref** *Pessoa* **scope** *pessoas*)

- A tabela *departamentos* pode ser criada como usual:

  create table *departamentos* **of** *Departamento*

- Em SQL:1999 pode-se omitir a declaração de **scope** na definição do tipo, passando-a para a criação da tabela, da seguinte forma:
  create table *departamentos* **of** *Departamento*
  (*presidente* **with options scope** *pessoas*)

Inicialização de referências no Oracle

- No Oracle, para criar um tuplo com uma referência, cria-se primeiro o tuplo com uma referência **null**, actualizando depois a referência com a função **ref**(p):
- E.g. para criar um departamento com nome *DI* e presidente Luís Caires:

```
insert into departamentos
      values (`DI', null)
update departamentos
    set presidente = (select ref(p)
                          from pessoas as p
                          where nome=`Luís Caires')
    where nome = `DI'
```

Inicialização de referências no SQL:1999

- O SQL:1999 não suporta a função **ref**(). Em vez disso exige a declaração de um atributo especial para guardar o identificador dum objecto
- O atributo com o identificador é declarado usando **ref is** na criação da tabela:

  create table *pessoas* **of** *Pessoa*
  ref is *oid* **system generated**

  - Aqui, *oid* é um nome dum atributo.
- Para obter a referência para um tuplo usa-se esse atributos. Por exemplo:

  select *p.oid*
  (em vez de **select ref**(*p*))

Geração de identificadores

- O SQL:1999 permite que os identificadores sejam gerados pelo utilizador
  - O tipo do identificador tem que ser declarado como para qualquer outro atributo da tabela
  - E.g.

```
create type Pessoa
        (nome varchar(20)
        morada varchar(20))
        ref using varchar(20);

create table pessoas of Pessoa
  ref is oid user generated;
```

- Quando se cria um tuplo, tem que se dizer qual o valor do identificador (que é sempre dado como primeiro atributo):

```
insert into pessoas values
  ('01284567', 'Pedro', `Lisboa');
```

Geração de identificadores (Cont.)

- Pode depois usar-se o valor inserido como referência, ao inserir tuplos noutras tabelas
  - Evita pergunta para saber qual o identificador:

  E.g. **insert into** *departamentos*
  values(`DI', `02184567')
- Também é possível usar uma chave primária como identificador. Para isso usa-se **ref from**, e declara-se a referência como sendo **derived**

```
create type Pessoa
        (nome varchar(20) primary key,
        morada varchar(20))
        ref from(nome)

create table pessoas of Pessoa
        ref is oid derived
```

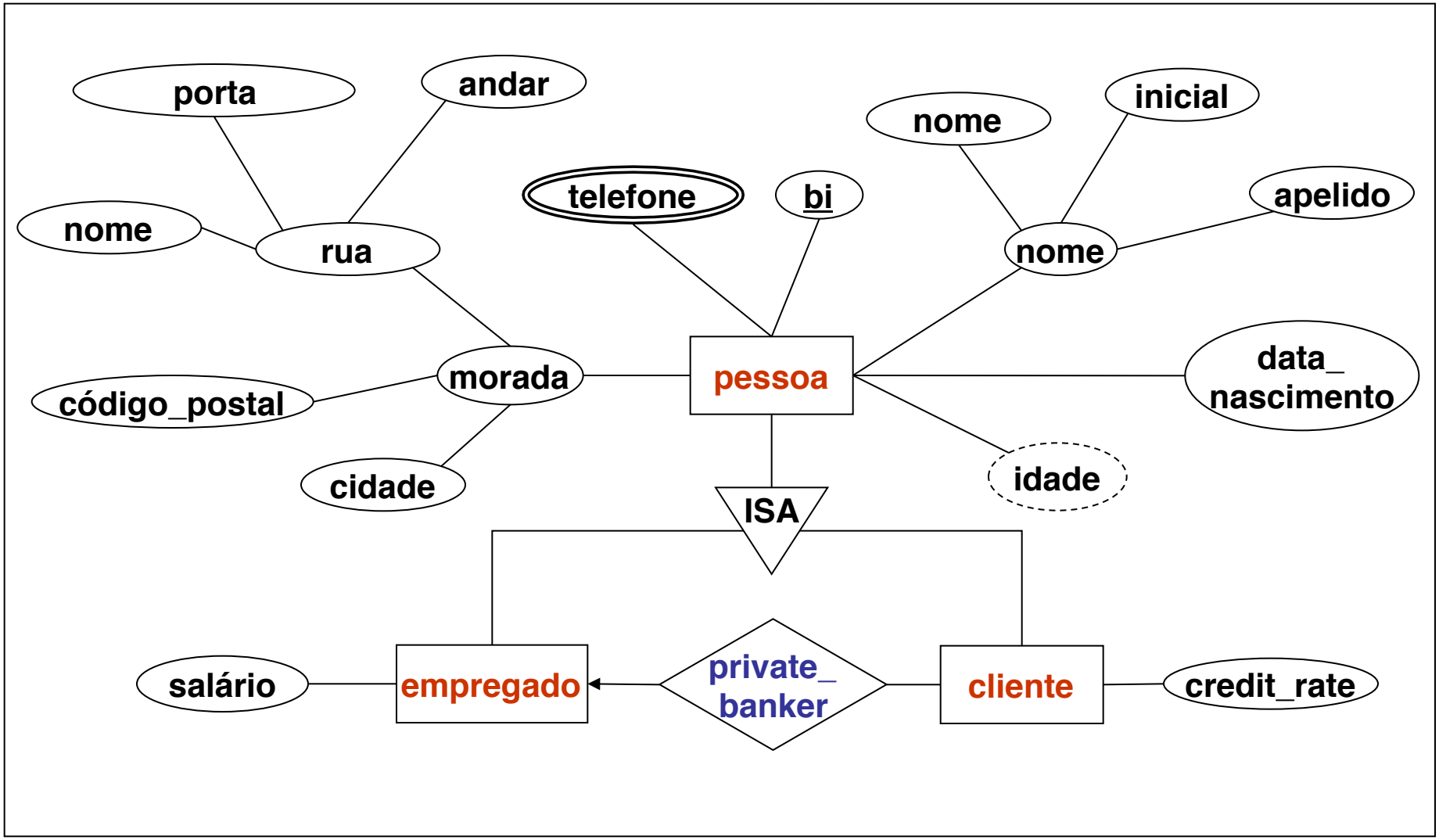
Path Expressions

- Quais os nomes e moradas de todos os presidentes de departamentos?

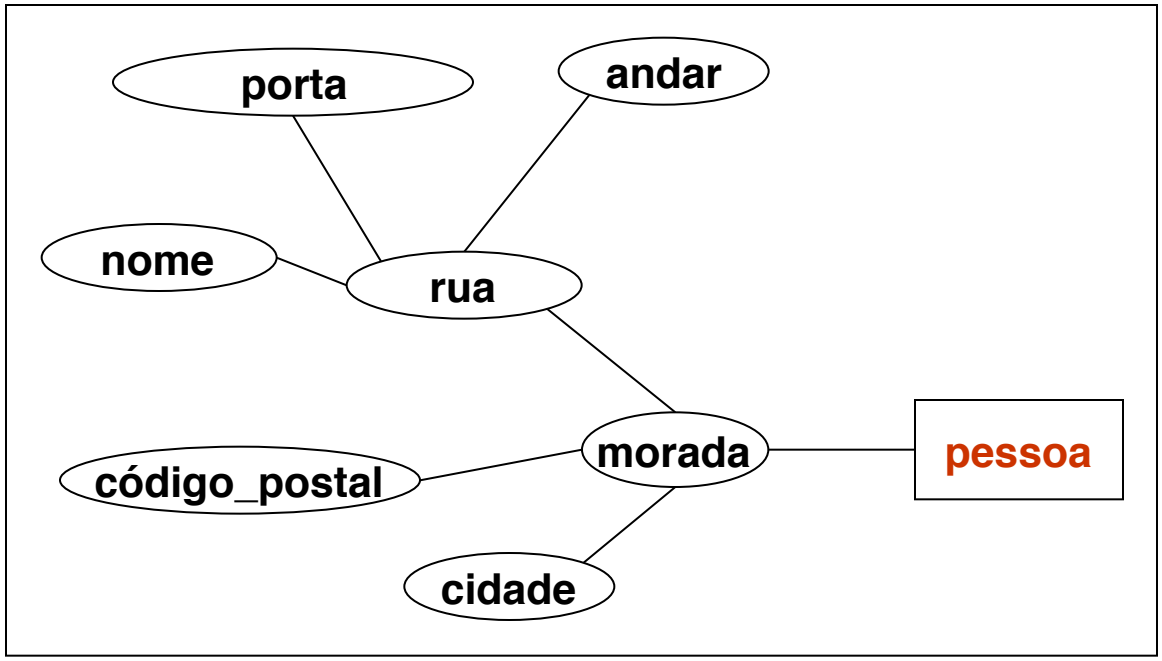
  select *presidente*–>*nome*, *presidente*–>*morada*
  from *departamentos*

- Uma expressão como "presidente–>nome" chama-se uma **path expression**
- No Oracle usa-se: "DEREF(presidente).nome"
- O uso de referências e *path expressions* evita operações de junção
  - Se *presidente* não fosse uma referência, para obter a morada era necessário fazer a junção de *departamentos* com *pessoas*
  - Facilita bastante a formulação de perguntas

Exemplo


porta
andar
nome
inicial
telefone
bi
apelido
nome
rua
nome
data_
nascimento
morada
pessoa
código_postal
cidade
idade
ISA
salário
empregado
private_
banker
cliente
credit_rate

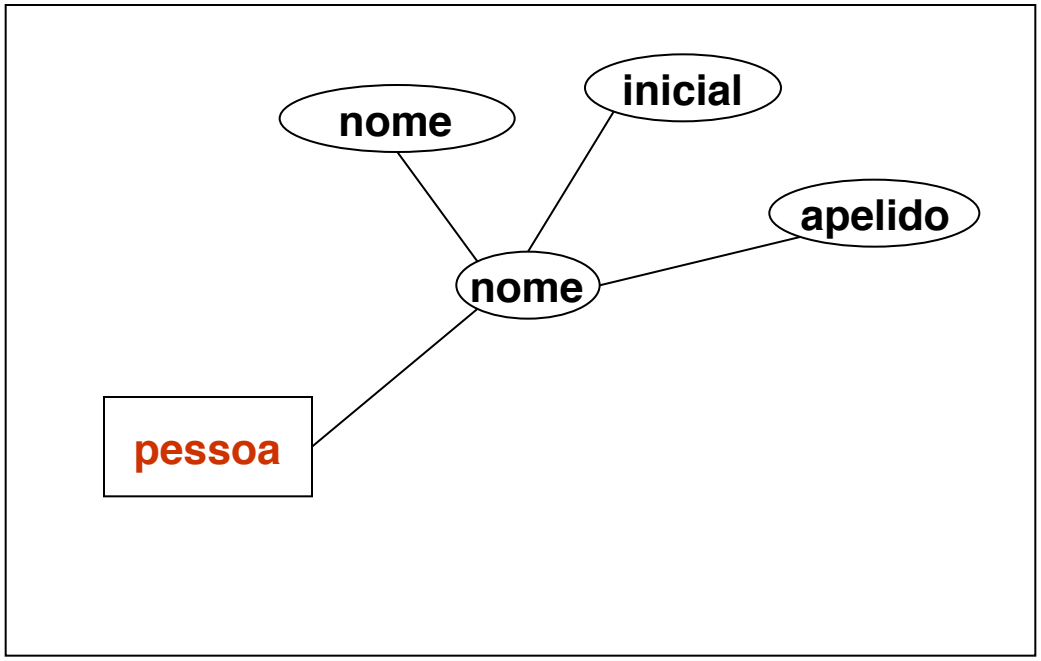
Exemplo



create type Rua **as**(
nome **varchar(15),**
porta **varchar(4),**
andar **varchar(7))**
final

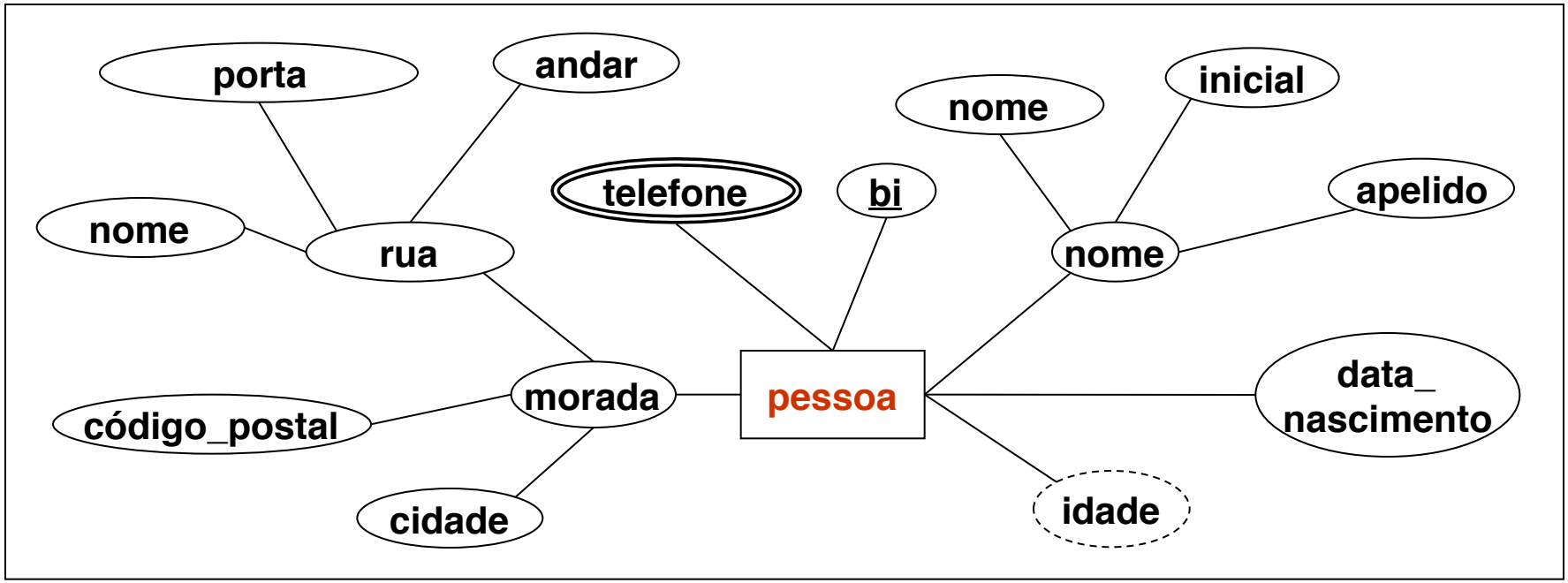
create type Morada **as**(
rua Rua,
cidade **varchar(15)**,
codigo_postal **char(8)**)
final

Exemplo




```
create type Nome as(
    nome varchar(15),
    inicial char,
    apelido varchar(15))
    final
```

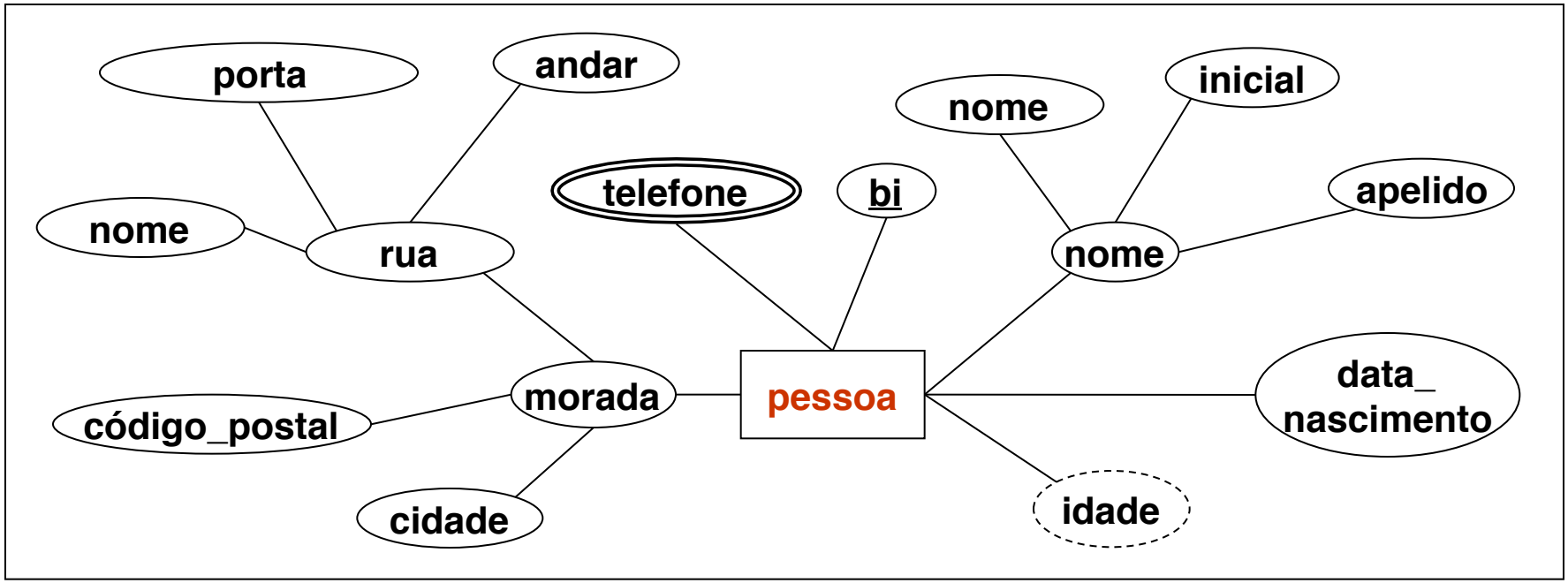
Exemplo




```
create type Pessoa as(
    nome Nome,
    bi number(10) primary key,
    morada Morada,
    telefone char(9) multiset,
    data_nascimento date)
    not final
```

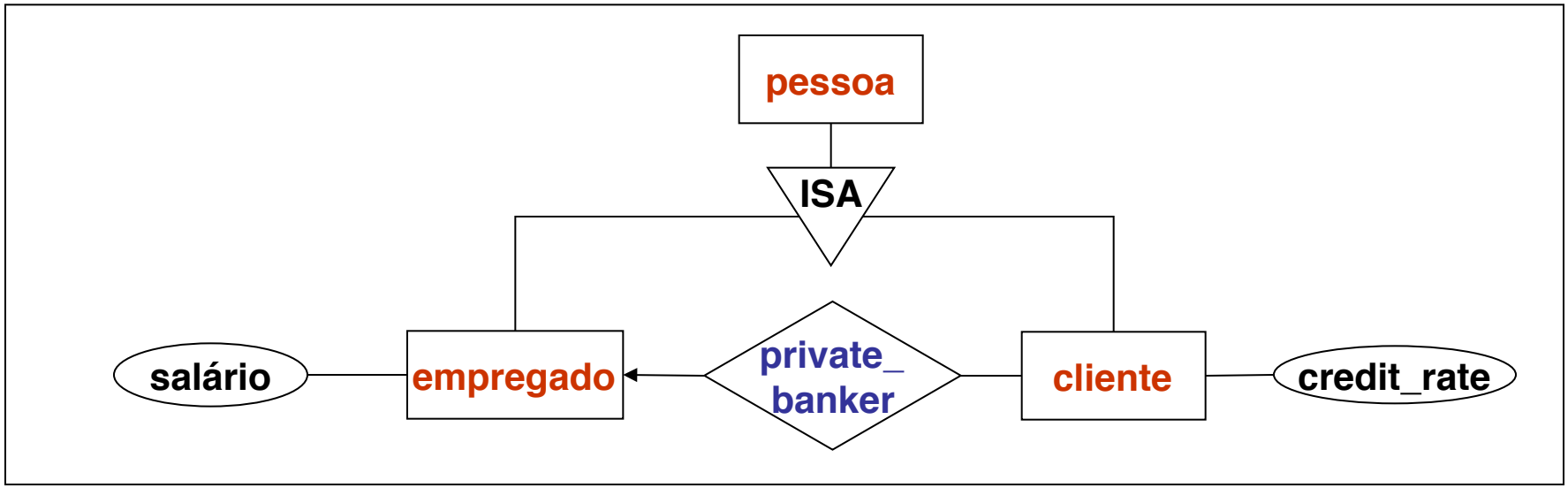
```
method idade(current_date date)
returns interval year
```

Exemplo




```
create instance method idade(current_date date)
    returns interval year
    for Pessoa
begin
    return current_date - self.data_nascimento;
end
```

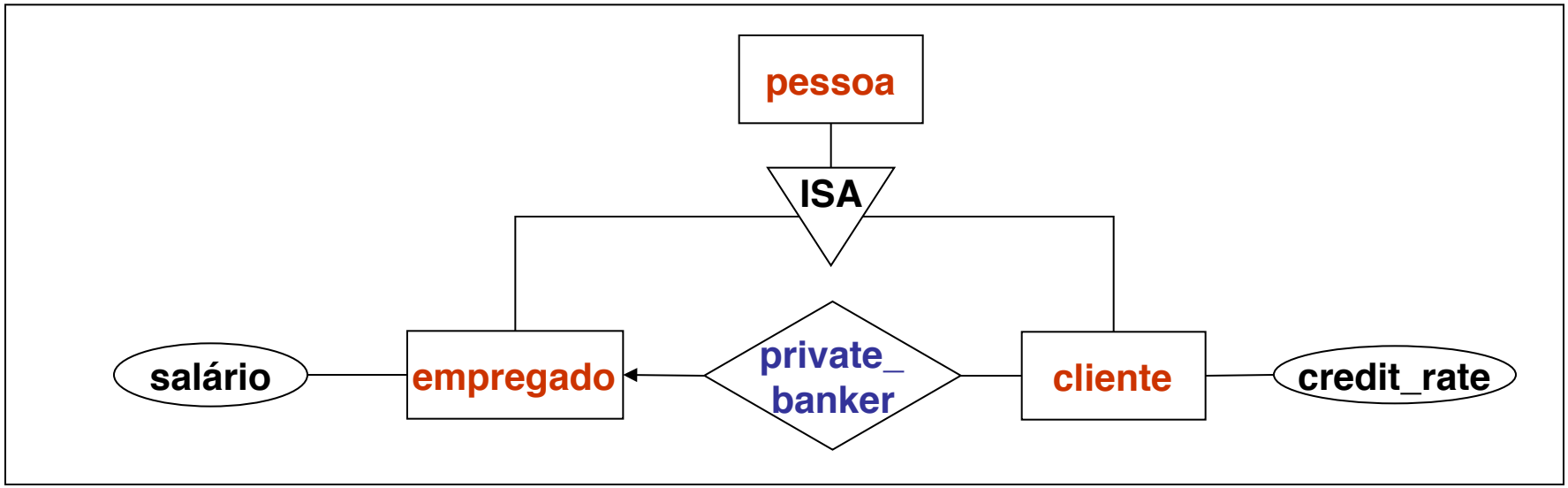
Exemplo



create type Empregado **under** Pessoa(
salario **number(10))**
final

create type Cliente **under** Pessoa(
credit_rate **char(1**),
private_banker **ref(**Empregado**)**)
final

Exemplo



create table pessoas **of** Pessoa

create table empregados **of** Empregado
under pessoas
ref is pessoa_id **system generated**
%(podia ser **user generated** ou **derived**)

create table clientes **of** Cliente
under pessoas
(private_banker **with options scope** empregados)

Modelo de Dados Objecto-Relacional

- **Colecções de dados (relações imbricadas)**: Permitir relações em todo o local onde antes se permitiam valor atómicos — relações dentro de relações
- **Tipos estruturados**: Permitir tipos com atributos compostos
- **Encapsulamento de dados**: Permitir a separação da estrutura dos dados do seu acesso — métodos associados aos tipos de dados
- **Herança**: Permitir a implementação directa de especializações
- **Identidade de Objectos e tipos de Referência**: Permitir a noção de identidade e referência de um objecto, independentemente do valor dos seus atributos.

Modelo de Dados Objecto-Relacional

- Os objectos:
  - Facilitam a modelação de entidades complexas.
  - Facilitam a reutilização, levando a um desenvolvimento de aplicações mais rápido.
  - Facilitam a compreensão e manutenção do software.
  - O suporte nativo de objectos por parte de uma SGBD permite o aceso directo às estruturas de dados usadas pelas aplicações, não necessitando de um nível de mapeamento.
  - Os objectos permitem o encapsulamento de operações juntamente com os dados (e.g. métodos para obter o valor de atributos derivados, consultas habituais, etc).
  - Os objectos permitem facilmente representar relações do tipo todo/parte.

Usos do modelo

- Estes conceitos de objectos podem ser usados de diversas formas:
  - Usar apenas como ferramenta na fase de design, e na implementação usa-se, e.g., uma base de dados relacional
    - Semelhante ao uso que fizemos de diagramas ER (que depois se passavam para conjunto de relações)
  - Incorporar as noções de objectos no sistema que manipula a base de dados:
    - **Sistemas objecto-relacional** – adiciona tipos complexos e noções de objectos a linguagem relacional.
    - **Linguagens persistentes** – generalizam (com conceitos de persistência e colecções) linguagens de programação object-oriented para permitir lidar com bases de dados.

Linguagens Persistentes

- Permitem criar e armazenar objectos em bases de dados , e usá-los directamente a partir da linguagem de programação (object-oriented, claro!)
  - Permite que os dados sejam manipulados directamente a partir do programa
- Programas em que os dados ficam de uma sessão para outra.
  - sendo isto feito de forma transparente para o utilizador
- Desvantagens:
  - Dada a expressividade das linguagens, é fácil cair em erros (de programação) que violam consistência dos dados.
  - Tornam muito difícil (senão impossível) optimização de consultas.
  - Não suportam formulação declarativa de perguntas (como acontece em bases de dados relacionais)

Persistência de Objectos

- Há várias abordagens para tornar os objectos persistentes
  - Persistência por Classe – declara-se que todos os objectos duma classe são persistentes; simples mas inflexível.
  - Persistência por Criação – estender a sintaxe de criação de objectos, para se poder especificar se são ou não persistentes.
  - Persistência por Marcação – todo o objecto que se pretende que persista para além da execução do programa, é marcado antes do programa terminar.
  - Persistência por *Reachability* – declara-se uma raiz de objectos persistentes; os objectos que persistem são aqueles que se conseguem atingir (directa ou indirectamente) a partir dessa raiz.
    - Facilita a vida ao programador, mas põe um overhead grande na base de dados
    - Semelhante a *garbage collection* usado em Java

Identidade de Objectos

- A cada objecto persistente é associado um identificador.
- Há vários níveis de permanência da identidade de objectos:
  - Intraprocedure – a identidade persiste apenas durante a execução dum procedimento
  - Intraprogram – a identidade persiste apenas durante a execução dum programa ou pergunta.
  - Interprogram – a identidade persiste duma execução dum programa para outra, mas pode mudar se a forma de armazenar os dados for alterada
  - Persistente – a identidade persiste entre execuções de programas e entre reorganizações de dados; este é o nível necessário para sistemas de bases de dados de objectos.

Linguagens Persistentes

- Há sistemas C++ e JAVA Persistentes já definidos e implementados:
  - C++
    - ODMG C++
    - ObjectStore
  - Java
    - Java Database Objects (JDO)

Comparação entre os vários sistemas

- **Sistemas relacionais**
  - Simplicidade no tipo de dados, linguagens declarativa para perguntas, boa segurança.
- **Linguagens persistentes**
  - Tipos de dados complexos, fácil integração com linguagens de programação, eficientes.
- **Sistemas Objecto-relacional**
  - Tipos de dados complexos, linguagens declarativa para perguntas, boa segurança.
- Nota: Os sistemas reais, esbatem um pouco estas fronteiras.
  - E.g. linguagens persistentes que funcionem como *wrapper* sobre uma base de dados relacional, embora permitam tipos de dados complexos e facilidade de integração com linguagens imperativas, nem sempre são muito eficientes…