TÉCNICO LISBOA

Bases de Dados

T09 - Derivar SQL de Esquemas Relacionais

Prof. Daniel Faria
Prof. Flávio Martins

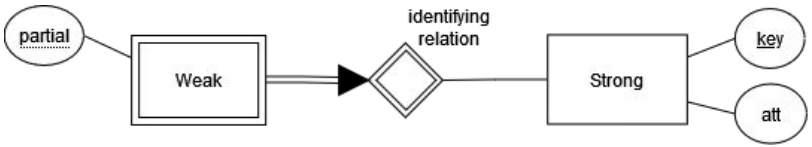
Sumário

- Conversão E-A–Relacional (Parte II)
  - Exercícios
- Derivar Bases de Dados SQL de Esquemas Relacionais
  - Exercícios

TÉCNICO LISBOA

Conversão E-A–Relacional (Parte II)

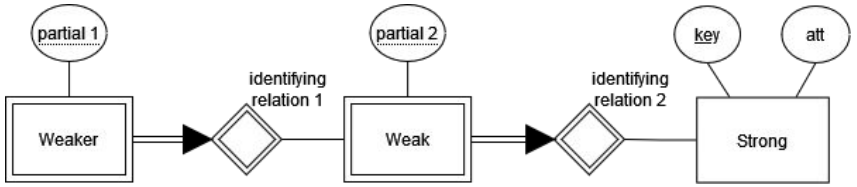
Entidades Fracas




```
Strong(key,att)
Weak(partial,key)
   key: FK(Entity)
IC: when a Strong is removed from
the database all the referencing
Weak must also be removed
```

- Semelhante a associações 1-N obrigatórias, mas a chave da entidade fraca é sempre composta pela sua chave parcial mais a chave da entidade forte
- É necessária uma restrição de integridade para forçar a eliminação da entidade fraca em caso de eliminação da entidade forte

Cadeias de Entidades Fracas




```
Strong(key,att)
Weak(partial_2,key)
    key: FK(Entity)
Weaker(partial1,partial_2,key)
    partial2,key: FK(Weak)
```

- Semelhante ao caso anterior, mas notar que a segunda entidade fraca (*Weaker*) tem uma chave estrangeira composta que aponta **apenas para a primeira** (*Weak*), não para a entidade forte

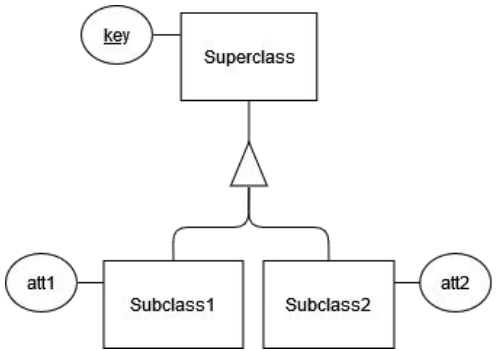
```
Weaker(partial1,partial_2,key)
    partial2: FK(Weak)
    key: FK(Strong)
```

Generalização/Especialização

- Opção 1: Mapear todas as entidades como relações
  - Usar chaves estrangeiras das subclasses para a superclasse
- Opção 2: Mapear apenas as subclasses como relações
- Opção 3: Mapear apenas a superclasse como relação, incorporando todos os atributos de todas as subclasses
- Complementar qualquer das opções com restrições de integridade para disjunção e cobertura

Generalização/Especialização

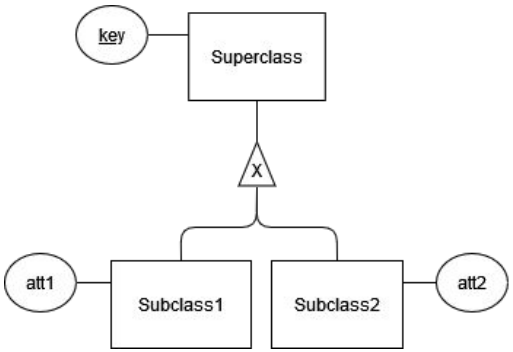
Opção 1: Aplicável em todos os casos




```
Superclass(key)
Subclass1(key,att1)
   key: FK(Superclass)
Subclass2(key,att2)
   key: FK(Superclass)
IC-1: when a Superclass is removed from
the database it must also be removed from
Subclass1 and/or Subclass2
```

Generalização/Especialização

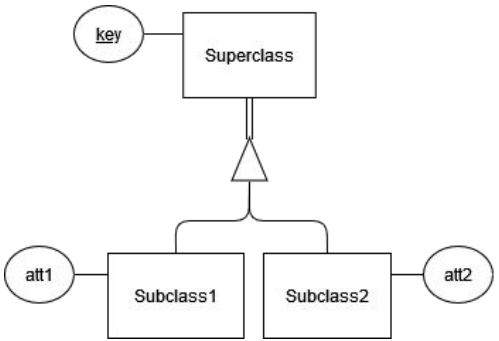
Opção 1: Aplicável em todos os casos




```
Superclass(key)
Subclass1(key,att1)
   key: FK(Superclass)
Subclass2(key,att2)
   key: FK(Superclass)
IC-1: when a Superclass is removed from
the database it must also be removed from
Subclass1 and/or Subclass2
IC-2: the same key cannot occur in
Subclass1 and Subclass2
```

Generalização/Especialização

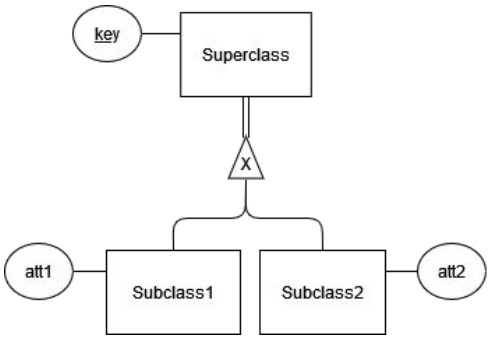
Opção 1: Aplicável em todos os casos




```
Superclass(key)
Subclass1(key,att1)
   key: FK(Superclass)
Subclass2(key,att2)
   key: FK(Superclass)
IC-1: when a Superclass is removed from
the database it must also be removed from
Subclass1 and/or Subclass2
IC-2: each key in Superclass must occur in
Subclass1 and/or Subclass2
```

Generalização/Especialização

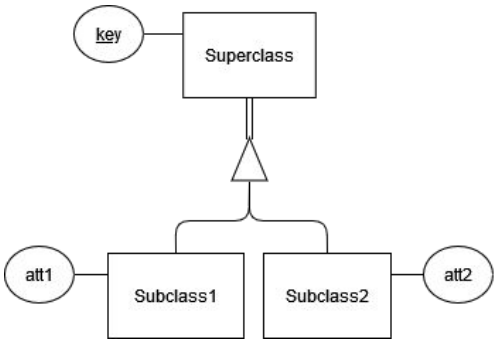
Opção 1: Aplicável em todos os casos




```
Superclass(key)
Subclass1(key,att1)
   key: FK(Superclass)
Subclass2(key,att2)
   key: FK(Superclass)
IC-1: when a Superclass is removed from
the database it must also be removed from
Subclass1 and/or Subclass2
IC-2: each key in Superclass must occur in
either Subclass1 or Subclass2 but not both
```

Generalização/Especialização

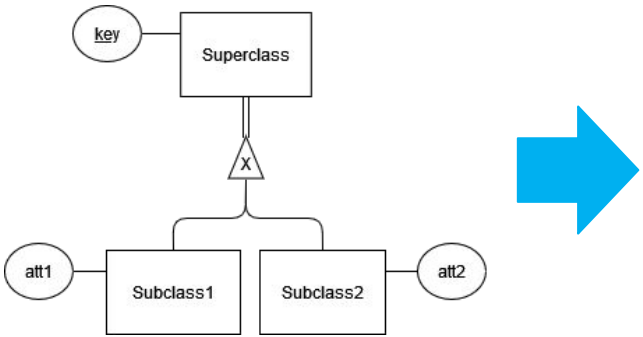
Opção 2: Aplicável se total



Subclass1(key,att1)
Subclass2(key,att2)
IC-1: when a Subclass1 is removed from the database, the Subclass2 with the same key must also be removed and vice-versa

Generalização/Especialização

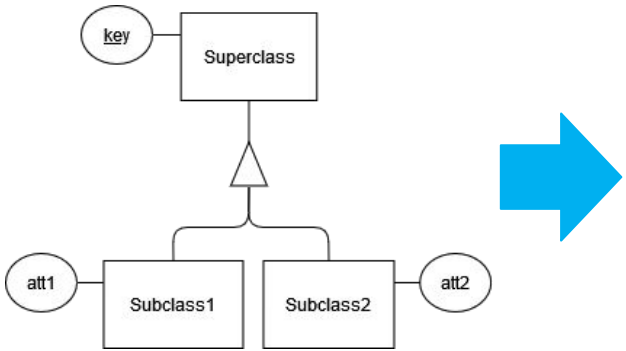
Opção 2: Aplicável se total




```
Subclass1(key,att1)
Subclass2(key,att2)
IC-1: the same key cannot occur in
Subclass1 and Subclass2
```

Generalização/Especialização

Opção 3: Aplicável em todos os casos, mas envolve NULLs



Superclass(key,att1,att2)

[Disjoint] IC-1: att1 and att2 cannot both be NOT NULL

[Total] IC-1: att1 and att2 cannot both be NULL

[Total Disjoint] IC1: one of att1 and att2 must be NULL and the other NOT NULL

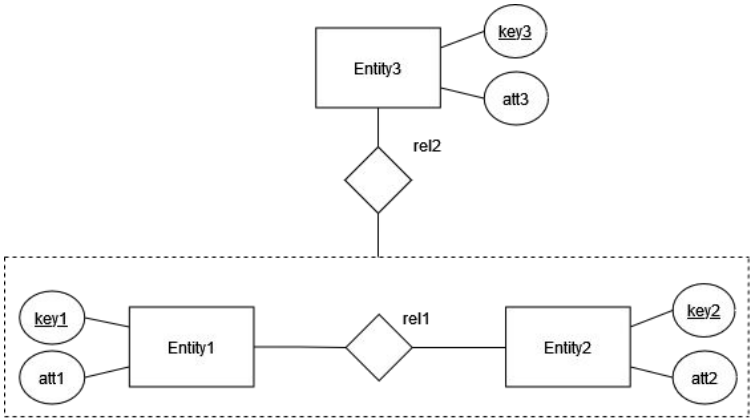
Generalização/Especialização

- Opção 1:
  - Menos eficiente, mas melhor opção em especialização parcial quando há relações ou vários atributos distintos
- Opção 2:
  - Mais eficiente para queries sobre subclasses específicas
- Opção 3:
  - Mais eficiente para queries sobre várias subclasses ao mesmo tempo
  - Leva a muitos NULLs se houver muitos atributos distintos
  - Mais difícil gerir relações distintas

Generalização/Especialização

- Considerações adicionais:
  - Se as especializações não têm atributos ou relações distintos mas são relevantes para o domínio, uma opção (não canónica) é captura-las com um atributo categórico
    - E.g. adicionar atributo “espécie” à relação “Animal de Estimação” em vez de ter especialização em Cão, Gato, etc
  - Se o único motivo para a especialização é a participação em associações distintas, a opção 3 é mais “limpa”: conseguimos distinguir as especializações pela sua presença nas relações que representam essas associações

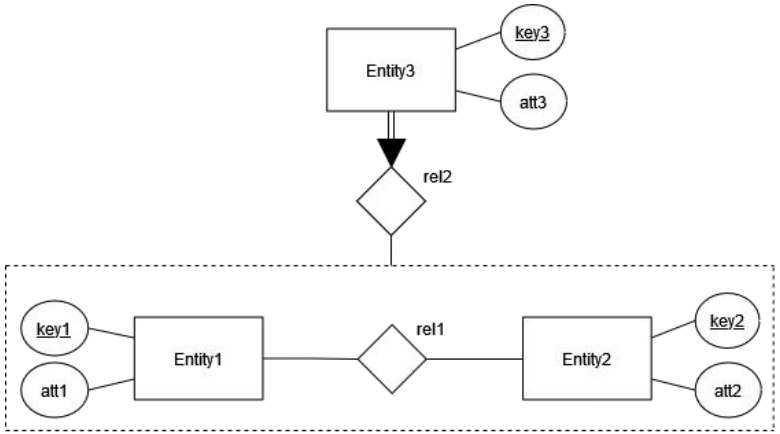
Agregação




```
Entity1(key1,att1)
Entity2(key2,att2)
rel1(key1,key2)
    key1: FK(Entity1)
    key2: FK(Entity2)
Entity3(key3,att3)
rel2(key1,key2,key3)
    key1,key2: FK(rel1)
    key3: FK(Entity3)
```

- Mapear primeiro o interior da agregação
- Idêntico a mapear associações
- Notar que a chave estrangeira de *rel2* é para *rel1*, não para as entidades

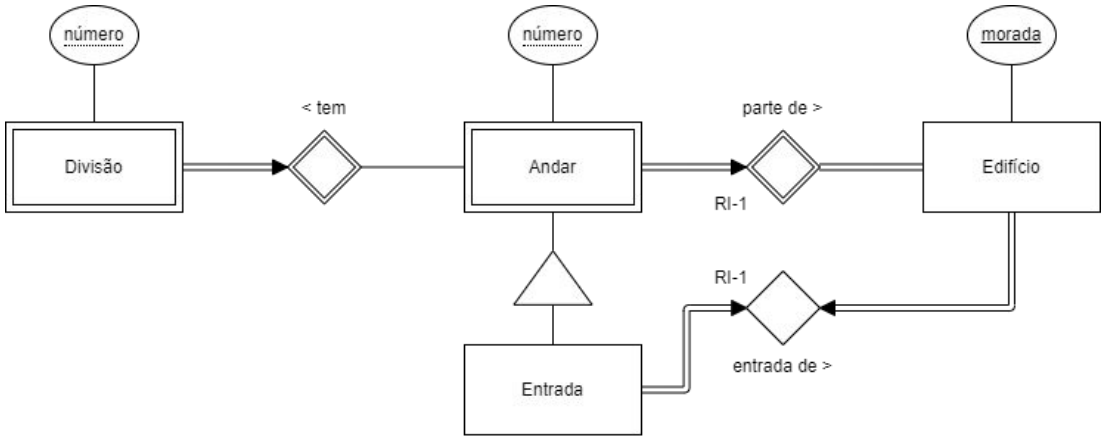
Agregação




```
Entity1(key1,att1)
Entity2(key2,att2)
rel1(key1,key2)
    key1: FK(Entity1)
    key2: FK(Entity2)
Entity3(key3,att3,key1,key2)
    key1,key2: FK(rel1) NOT NULL
```

- Aplicam-se os mesmo princípios do que a mapear associações
- Mas novamente, a chave estrangeira de *Entity3* é para *rel1*, não para as entidades

Exercício: Converter para Relacional

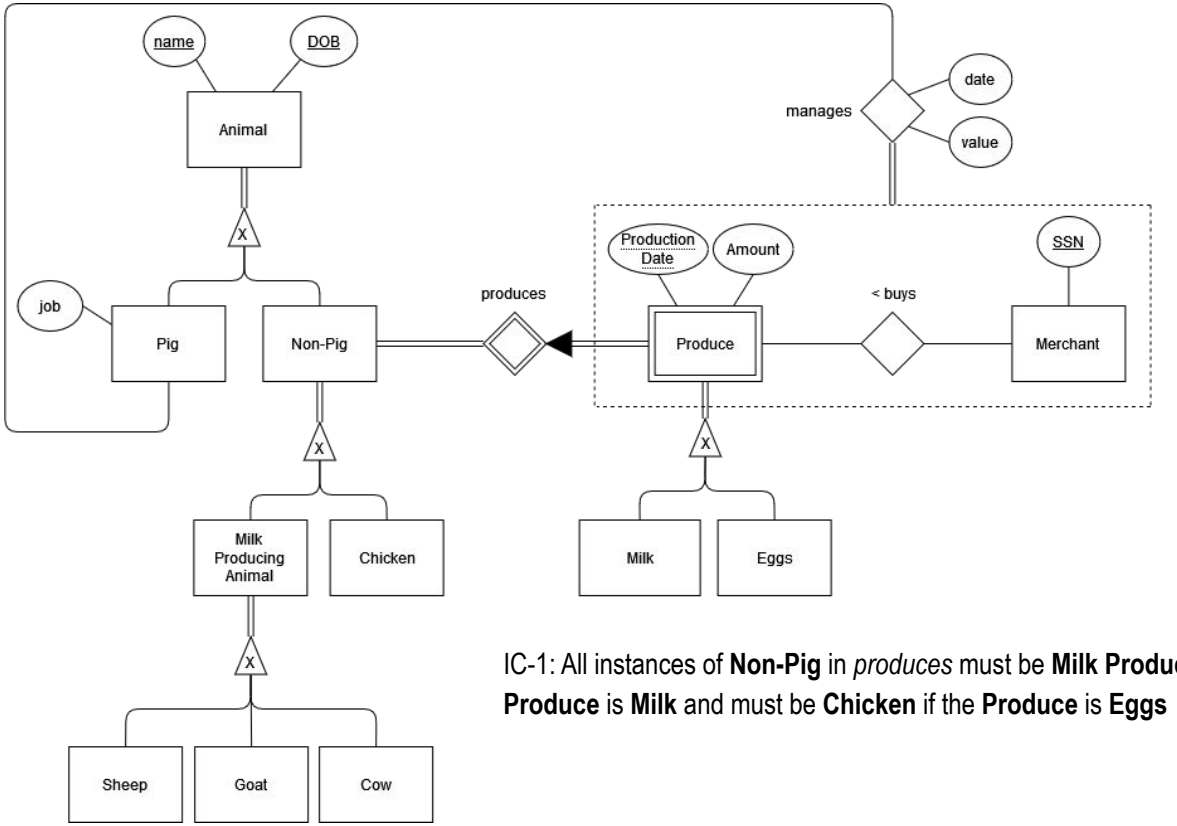


RI-1: Um Andar que é entrada de um edifício tem de ser também parte de esse edfiício

Solução

Edifício(morada)
Andar(número,morada)
    morada: FK(Edifício)
Entrada(número,morada)
    número,morada: FK(Andar)
Divisão(número_div,número,morada)
    número,morada: FK(Andar)
IC-1: todas as instâncias de Edifício têm de estar em Andar
IC-2: todas as instâncias de Edifício têm de estar em Entrada

Exercício: Converter para Relacional



IC-1: All instances of **Non-Pig** in *produces* must be **Milk Producing Animal** if the **Produce** is **Milk** and must be **Chicken** if the **Produce** is **Eggs**

TÉCNICO LISBOA

Solução Tradicional

```
Animal(name,DOB)
Pig(name,DOB,job)
   name,DOB: FK(Animal)
Non-Pig(name,DOB)
   name,DOB: FK(Animal)
Chicken(name,DOB)
   name,DOB: FK(Non-Pig)
Milk-Producer(name,DOB)
   name,DOB: FK(Non-Pig)
…
```

```
Produce(name,DOB,production date,amount)
   name,DOB: FK(Non-Pig)

Eggs(name,DOB,production date)
   name,DOB,production date: FK(Produce)

Milk(name,DOB,production date)
   name,DOB,production date: FK(Produce)

Buys(SSN,name,DOB,production date)
   SSN: FK(Merchant)
   name,DOB,production date: FK(Produce)

Manages(name1,DOB1,SSN,name2,DOB2,production
date, date, value)
   name1,DOB1: FK(Pig)
   SSN,name2,DOB2,production date: FK(Buys)
```

+ integrity constraints

Solução Tradicional

- Problema:
  - *Eggs* e *Milk* têm de referenciar *Produce* para serem subclasses, o que significa que não temos como referenciar *Chicken* e *Milk-Producer* para capturar a semântica do seu relacionamento
    - Não ganhamos grande coisa com a especialização em *Chicken* e *Milk-Producer* ou com a especialização em *Eggs* e *Milk*, e perdemos desempenho (inserir dados em duplicado com verificação de FKs)
- Alternativa:
  - Representar a espécie e o tipo de produto como atributos

Solução Prática

```
Animal(name,DOB)
Pig(name,DOB,job)
    name,DOB: FK(Animal)
Non-Pig(name,DOB,species)
    name,DOB: FK(Animal)
Produce(name,DOB,production date,amount,type)
    name,DOB: FK(Non-Pig)
Merchant(SSN)
Buys(SSN,name,DOB,production date)
    SSN: FK(Merchant)
    name,DOB,production date: FK(Produce)
Manages(name1,DOB1,SSN,name2,DOB2,production date, date, value)
    name1,DOB1: FK(Pig)
    SSN,name2,DOB2,production date: FK(Buys)
```

Solução Prática

IC-1: all instances of Animal must be either Pig or Non-Pig but not both

IC-2: species must be one of ‘cow’, ‘goat’, ‘sheep’, ‘chicken’

IC-3: type must be one of ‘eggs’, ‘milk’

IC-4: If type is ‘eggs’ in Produce then the species of Non-Pig must be ‘chicken’, and if the type is ‘milk’ the species must not be ‘chicken’

IC-5: All instances of Buys must be present in Manages

TÉCNICO LISBOA

Derivar Bases de Dados SQL de Esquemas Relacionais

Criar Tabelas SQL

```
CREATE TABLE table_name (
    column_name data_type [column_constraint],
    …
    [table_constraint],
    …
);
```

TÉCNICO LISBOA

Restrições de Coluna e Tabela

- Coluna:
  - **NOT NULL**
  - **PRIMARY KEY**
  - **UNIQUE**
  - **REFERENCES** *reftable* [(*refcolumn*)]
  - **CHECK** *expression*
- Tabela:
  - **PRIMARY KEY** (*column_name* [, ...])
  - **UNIQUE** (*column_name* [, ...])
  - **FOREIGN KEY** (*column_name* [, ...]) **REFERENCES** *reftable* [(*refcolumn*)]
  - **CHECK** *expression*

Mapeamento Modelo Relacional → SQL

- Nome da relação → nome da tabela
- Para cada atributo
  - Nome de atributo → nome de atributo
    ??? → tipo de dados do atributo
- Chave primária → declaração de PRIMARY KEY (coluna ou tabela)
- UNIQUE → declaração de UNIQUE (coluna ou tabela)
- NOT NULL → declaração de NOT NULL (coluna)
- FK → declaração de REFERENCES / FOREIGN KEY (coluna / tabela)
- ICs (alguns) → declarações de CHECK (apenas valores de um tuplo a inserir)

Mapeamento Modelo Relacional → SQL

Exemplo:

Consulta(médico,paciente,gabinete, hora_início,hora_fim)
médico: FK(Médico.id)
paciente: FK(Paciente.id) NOT NULL
UNIQUE(médico,hora_fim)
UNIQUE(paciente,hora_início)
UNIQUE(paciente,hora_fim)
UNIQUE(gabinete,hora_início)
UNIQUE(gabinete,hora_fim)
IC-1: A hora_início tem de ser inferior à hora_fim
IC-2: O gabinete tem de ser um número inteiro entre 1 e 12
IC-3: Não pode haver uma consulta com data_início entre a data_início e a data_fim de outra consulta que tenha o mesmo médico ou o mesmo paciente ou o mesmo gabinete

Mapeamento Modelo Relacional → SQL

```
CREATE TABLE consulta(
    medico      INTEGER REFERENCES Medico(id),
    paciente    INTEGER NOT NULL REFERENCES Paciente(id),
    gabinete    INTEGER NOT NULL CHECK gabinete BETWEEN 1 AND 12,
    hora_inicio TIMESTAMP,
    hora_fim    TIMESTAMP NOT NULL,
    PRIMARY KEY (medico, hora_inicio),
    UNIQUE (medico, hora_fim),
    UNIQUE (paciente, hora_inicio),
    UNIQUE (paciente, hora_fim),
    UNIQUE (gabinete, hora_inicio),
    UNIQUE (gabinete, hora_fim),
    CHECK hora_fim > hora_inicio);
```

Mapeamento Modelo Relacional → SQL

`IC-3: Não pode haver uma consulta com data_início entre a data_início e a data_fim de outra consulta que tenha o mesmo médico ou o mesmo paciente ou o mesmo gabinete`

- Não conseguimos verificar com CHECK
    - CHECK apenas permite fazer validações de valores do tuplo que estamos a inserir, não conseguimos comparar com valores já existentes na tabela
- Apenas pode ser verificado com um Trigger

Exercício: Converter para SQL

```
Animal(name,DOB)

Pig(name,DOB,job)
    name,DOB: FK(Animal)

Non-Pig(name,DOB,species)
    name,DOB: FK(Animal)

Produce(name,DOB,production date,amount,type)
    name,DOB: FK(Non-Pig)

Merchant(SSN)

Buys(SSN,name,DOB,production date)
    SSN: FK(Merchant)
    name,DOB,production date: FK(Produce)

Manages(name1,DOB1,SSN,name2,DOB2,production date, date, value)
    name1,DOB1: FK(Pig)
    SSN,name2,DOB2,production date: FK(Buys)
```

Exercício: Converter para SQL

IC-1: all instances of Animal must be either Pig or Non-Pig but not both

IC-2: species must be one of ‘cow’, ‘goat’, ‘sheep’, ‘chicken’

IC-3: type must be one of ‘eggs’, ‘milk’

IC-4: If type is ‘eggs’ in Produce then the species of Non-Pig must be ‘chicken’, and if the type is ‘milk’ the species must not be ‘chicken’

IC-5: All instances of Buys must be present in Manages

Solução

```
CREATE TABLE merchant(
    SSN NUMERIC(12) PRIMARY KEY);

CREATE TABLE animal(
    name VARCHAR(80),
    DOB DATE,
    PRIMARY KEY(name,DOB));

CREATE TABLE pig(
    name VARCHAR(80),
    DOB DATE,
    job  VARCHAR(80) NOT NULL,
    PRIMARY KEY(name,DOB),
    FOREIGN KEY(name,DOB) REFERENCES animal);
```

Solução

```
CREATE TABLE nonpig(
    name VARCHAR(80),
    DOB DATE,
    species VARCHAR(80) NOT NULL CHECK
        (species IN ('cow','goat','sheep','chicken')),
    PRIMARY KEY(name,DOB),
    FOREIGN KEY(name,DOB) REFERENCES animal);

CREATE TABLE produce(
    name VARCHAR(80),
    DOB DATE,
    production_date DATE,
    amount  FLOAT NOT NULL,
    type CHAR(4) NOT NULL CHECK type IN ('eggs','milk'),
    PRIMARY KEY(name,DOB,production_date),
    FOREIGN KEY(name,DOB) REFERENCES non-pig);
```

Solução

```
CREATE TABLE buys(
    SSN NUMERIC(8) REFERENCES merchant,
    name VARCHAR(80),
    DOB DATE,
    production_date DATE,
    PRIMARY KEY(SSN,name,DOB,production_date),
    FOREIGN KEY(name,DOB,production_date) REFERENCES produce);

CREATE TABLE manages(
    name1 VARCHAR(80),
    DOB1 DATE,
    SSN NUMERIC(8) REFERENCES merchant,
    name2 VARCHAR(80),
    DOB2 DATE,
    production_date DATE,
    PRIMARY KEY(name1,DOB1,SSN,name2,DOB2,production_date),
    FOREIGN KEY(name1,DOB1) REFERENCES pig,
    FOREIGN KEY(SSN,name2,DOB2,production_date) REFERENCES buys);
```

Solução

IC-4: If type is 'eggs' in Produce then the species of Non-Pig must be 'chicken', and if the type is 'milk' the species must not be 'chicken'

- Não conseguimos verificar com um CHECK porque são atributos de tabelas diferentes
  - Podíamos importar "species" em "Produce" (acrescentando-o à chave estrangeira que já tem para "Non-Pig") para permitir usar CHECK, mas é uma duplicação de dados desnecessária sem ganho claro de desempenho

Comentário

- A solução anterior é um bom exemplo de porque a prática *default* é criar chaves primárias numéricas (auto-incrementais) para todas as tabelas
    - O acréscimo ao volume de dados da tabela em que é acrescentado é largamente compensado pelo que poupa nas tabelas que a referenciam, por evitar duplicar chaves de tipos textuais ou chaves compostas
- No entanto, essa prática não será contemplada nesta fase da disciplina
    - Devem declarar as chaves existentes nos dados / modeladas no diagrama E-A

TÉCNICO LISBOA