Apontamentos de BD

João Aragonez

Conteúdo

Capítulo 1

Conceitos Iniciais

1.1 Sistemas de informação

Definição 1 (Sistemas de Informação)**.** Consiste na área que estuda as atividades de pendor estratégico, operacional e de gestão subjacentes à recolha, processamento, armazenamento, distribuição e uso de informação e de tecnologias associadas, tanto pela sociedade como por organizações.
Também é comum definir SI como a interação entre tecnologia e processos de negócio, mais concretamente, a gestão de 3 componentes fundamentais: **dados, tecnologia e pessoas**.

Entre outros, menciona-se os seguintes tipos de sistemas de informação:

- *ERP* (*Enterprise Resource Planning*);
- SIG (Sistemas de Informação Geográfica);
- Sistemas de *office automation*;
- Sistemas de *Business Intelligence*;
- Sistemas Especialistas;
- *WWW* (*World Wide Web*).

1.2 Sistemas de Gestão de Bases de Dados (SGBD)

Definição 2 (Base de Dados)**.** Consiste em nada mais que conjuntos de dados interligados.

Definição 3 (Sistema de Gestão de Bases de Dados)**.** Consiste numa ferramenta de software desenhada para a manutenção e gestão de bases de dados

Dado que os sistemas operativos atuais se encontram munidos de um sistema de ficheiros, perfeitamente capazes de lidar com o armazenamento de informação, surge a seguinte questão: *porquê usar um SGBD?* A verdade é que os sistemas de informação apresentam necessidades comuns que não são cobertas por sistemas de ficheiros. Assim, os SGBD têm por objetivo realizar:

- Controlo de redundância;
- Segurança e controlo de acessos, dada a heterogeneidade de utilizadores e de dados;
- Persistência de dados;
- Oferecer múltiplas interfaces para diferentes tipos de utilizadores;
- Representar relações complexas;
- Assegurar constrangimentos de integridade sobre os dados;
- Realizar controlo de concorrência, por forma a manter os dados consistentes;
- Permitir que uma grande quantidade de iterrogações (*queries*) possam ser feitas sobre os dados sem necessidade de programação adicional;
- Garantir tolerância a faltas (e.g., realizando *backups*).

1.2.1 Vantagens dos SGBD's

- **Independência dos dados**: encapsulando o modo real de representação e armazenamento dos dados, os SGBD's disponibilizam uma visão abstrata dos dados.
- **Acesso Eficiente aos Dados**: os SGBD incorporam técnicas para armazenamento e recolha eficiente dos dados;
- **Integridade dos dados e segurança**: os SGBD garantem a aplicação de restrições de integridade no acesso e manipulação de dados;
- **Capacidade de administração dos dados**: é possível mudar a representação dos dados por forma a minimizar a redundância e melhorar o armazenamento de forma totalmente transparente ao utilizador;
- **Acesso Concorrente e Recuperação de Falhas**: existe suporte à concorrência no acesso aos dados, garantido um efeito semelhante a um acesso sequencial;
- **Redução do tempo de desenvolvimento de aplicações**: disponibiliza uma interface de alto nível para os dados e funções de acesso comuns, sendo para além disso uma componente da aplicação que não necessita de ser verificada.

1.2.2 Desvantagens dos SGBD's

- **Overhead demasiado elevado**: requer investimento em hardware, software e formação no uso destes sistemas;
- **Tratamento demasiado geral**: Dependendo da aplicação, os mecanismos de segurança, controlo de concorrência, integridade e de recuperação de faltas podem não ser suficientes;
- **Desadequados a sistemas com requisitos de tempo-real**;
- **Desadequados a bases de dados simples/imutáveis ou sem concorrência de acessos**;
- **Desadequados a certos tipos de dados, como texto**.

1.3 Modelos e Níveis de Abstração nos SI

Num SGBD, os dados podem ser descritos segundo diversos modelos, que correspondem a diferentes níveis de abstração acerca da sua representação/armezenamento:

- **Modelo Conceptual** (ou esquema externo), que descreve como os utilizadores vêm os dados. Permite particulizar o acesso aos dados através de **Vistas** - conjuntos de registos visíveis para grupos específicos de utilizadores e apenas computados quando necessário (i.e., não são explicitamente armazenados). Este nível permite **independência dos dados lógicos**, pois alterações ao esquema lógico requerem unicamente redefinição de vistas, pelo que o utilizador não se dará conta de eventuais extensões e modificações das estruturas de dados.
- **Modelo Lógico** (ou esquema conceptual), que corresponde à estrutura lógica dos dados (e.g., relações existentes no modelo relacional). Este nível permite **independência dos dados físicos**, pois a organização física nada influi sobre o esquema lógico dos dados.
- **Modelo Interno** (ou esquema físico), que especifica os detalhes de armazenamento das relações (e.g., definição de tipos de ficheiros a utilizar e de índices).

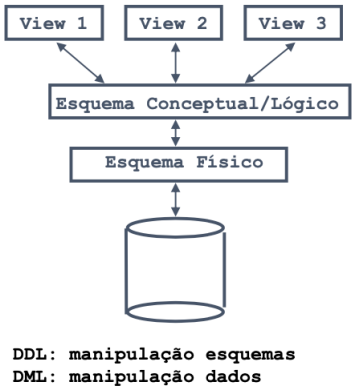


Figura 1.1: Modelo *ANSI/SPARC*

1.4 Modelos de Dados

Definição 4 (Modelo de Dados)**.** Coleção de conceitos para descrever dados, relacionamentos, semântica de dados e restrições.

Definição 5 (Esquema)**.** Descrição de uma coleção específica de dados à luz de um dado modelo de dados (i.e., o resultado da aplicação de um modelo de dados um conjunto de dados específico).

Entre outros modelos de dados, destacam-se o **Modelo Relacional**, o **Modelo Entidade-Associação**, o **Modelo Baseado em Objetos**, **Modelos de Dados Semi-Estruturados** (como *XML/JSON*), ou os **Modelos em Rede e Hierárquicos** (não usados atualmente).
Contudo, o modelo de dados mais amplamente difuso nos SGDB é o **modelo relacional**, cujos conceitos fundamentais são a **relação** (i.e., um tuplo de atributos) e o **esquema**, que corresponde à especificação do nome da relação e do nome e tipo dos seus atributos.
Numa fase mais inicial do desenvolvimento de bases de dados, podem-se usar **Modelos Semânticos de Dados**, passíveis de serem diretamente traduzidos para o modelo relacional. O exemplo mais paradigmático destes modelos é o **Modelo Entidade-Associação**.

1.5 Arquitetura dos SGBD

As arquiteturas dos SGBD procuram, por um lado, maximizar a **eficiência e escalabilidade**, mais concretamente, acelerando as interrogações sobre os dados. A figura abaixo exibe as fases que compõem o processamento de uma *query*: **análise e tradução**, **otimização** e **avaliação**.

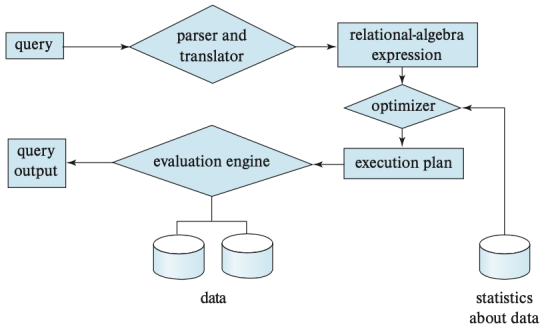


Figura 1.2: Processamento de uma *query*

Por outro lado, procuram maximizar a **concorrência e a robustez**, existindo um **gestor de transações** para lidar com questões de concorrência, bem como um **gestor de recuperação** e um **gestor de *locks***.

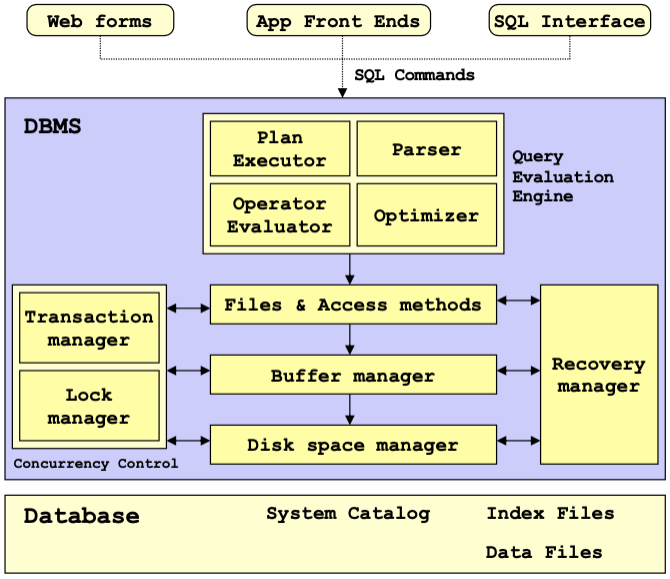


Figura 1.3: Arquitetura de um SGBD

1.6 Conceção de Bases de Dados

O processo de conceção de um base de dados incide inicialmente no **desenho lógico**, i.e., sobre o **esquema** a adotar. Para este decisão contribuem fatores associados ao **Negócio** (como determinar quais os atributos mais relevantes para o domínio em questão), bem como fatores de **Engenharia**, como definição de esquemas e distribuição dos atributos por estes.

1.7 Utilizadores de Bases de Dados

- **Implementadores de Bases de Dados;**
- **Utilizadores das aplicações;**
- **Programadores de aplicações** ao definirem o modelo lógico do sistema de informação;
- **DBA** (*Database Administrators*), que concebem e mantêm a base de dados, em termos de desenho físico e lógico, segurança e configuração dos mecanismos de disponibilidade e recuperação.

Capítulo 2

SQL

2.1 Visão Global da Linguagem *Query* SQL

A linguagem SQL tem várias partes:

- ***SQL Data-Definition Language (DDL)***: fornece comandos para definir esquemas relacionais, apagar relações e modificar esquemas relacionais. Inclui ainda comandos para especificação das restrições de integridade que os dados guardados na base de dados devem satisfazer - **Integridade**, comandos para definir vistas - **Definição de Vistas**, e comandos para especificar direitos de acesso às relações e vistas - **Autorização**.
- ***SQL Data-Manipulation Language (DML)***: fornece a capacidade de consultar informação da base de dados e de inserir em, remover de, e modificar tuplos na base de dados.
- **Controlo de Transações**: o SQL inclui comandos para especificar os pontos iniciais e finais de transações.
- **SQL Embutido e Dinâmico**: SQL embutido é a parte do SQL que é fixo e não pode ser mudado em *run-time*, enquanto que o SQL dinâmico permite acesso à base de dados em *run-time*.

2.2 Definição de Dados em SQL

O conjunto de relações numa base de dados são especificados usando uma linguagem *data-definition*. O SQL *DDL* permite a especificação de um conjunto de relações, bem como de informação sobre cada relação, incluindo:

- O **esquema** para cada relação;
- Os **tipos de valores** associados a cada atributo;
- As **restrições de integridade**;
- O conjunto de **índices** a manter em cada relação;
- A informação de **segurança** e **autorização** para cada relação;
- A estrutura de **armazenamento físico** de cada relação em disco.

2.2.1 Tipos Básicos

O *standard* SQL suporta uma variedade de tipos, incluindo:

- **char**(n): uma cadeia de texto de comprimento fixo de n;
- **varchar**(n): uma cadeia de texto de comprimento variável até n;
- **int**: um inteiro (é equivalente escrever **integer**) - depende da máquina em que opera, mas geralmente está no intervalo $[-2^{31}, 2^{31} - 1]$;
- **smallint**: um inteiro pequeno - depende da máquina em que opera, mas geralmente está no intervalo $[-2^{15}, 2^{15} - 1]$;

- **numeric**(p, d): um número de vírgula fixa com precisão especificada pelo utilizador - o número consiste de **p** dígitos (mais o sinal) e **d** desses **p** dígitos são à direita da vírgula. **numeric**(3,1) permite -44.5 ser guardado, mas não permite 444.5, nem 0.32.
- **real, double precision**: números de vírgula flutuante e número de vírgula flutuante com precisão de *double* - depende da máquina em que opera.
- **float**(n): um número de vírgula flutuante com uma precisão de **pelo menos** n dígitos.

2.2.2 Definição Básica de Esquemas

Definimos uma relação SQL usando o comando **CREATE TABLE**. O seguinte comando cria a relação *departamento* na base de dados:

```
CREATE TABLE departamento
      (nome_dept VARCHAR(20),
       edificio VARCHAR(15),
       orcamento NUMERIC(12, 2)),
       PRIMARY KEY (nome_dept));
```

A forma geral do comando **CREATE TABLE** é:

```
CREATE TABLE r
      (A_1 D_1,
       A_2 D_2,
       ...,
       A_n D_n,
       <restricao-integridade 1>,
       ...,
       <restricao-integridade k>);
```

O SQL suporta uma variedade de **restrições de integridade**. Algumas delas são:

- **primary key** $(A_{j_1}, A_{j_2}, ..., A_{j_m})$: esta especificação diz que os atributos $A_{j_1}, A_{j_2}, ..., A_{j_m}$ formam a chave primária da relação - por definição estes devem ser **NOT NULL** e **UNIQUE**, i.e., nenhum tuplo pode ter um valor **null** como atributo de chave primária, e não existem 2 tuplos numa relação com atributos de chave primária iguais. Esta especificação é **opcional**.
- **foreign key** $(A_{k_1}, A_{k_1}, ..., A_{k_n})$ **references** s: esta especificação diz que os valores dos atributos $(A_{k_1}, A_{k_2}, ..., A_{k_n})$ para qualquer tuplo na relação devem corresponder aos atributos de chave primária de algum tuplo da relação s.
- **not null**: esta especificação aplica-se a um atributo e especifica que o valor **null** não é permitido nele, i.e., excluir o valor **null** do domínio do mesmo.

O SQL previne qualquer atualização à base de dados que viole uma restrição de integridade.
Para remover uma relação r de uma base de dados SQL usamos o comando **DROP TABLE *r***, que apaga toda a informação acerca da relação a apagar da base de dados.
Para removermos apenas os tuplos contidos numa relação r usamos o comando **DELETE FROM *r*** - de notar que não precisaríamos criar a relação de novo ao usar este comando, a relação apenas ficaria sem qualquer tipo de dados contida nela.
Para adicionarmos um atributo A de tipo D a uma relação r usamos o comando **ALTER TABLE *r* ADD *A D*** - de notar que a todos os tuplos é-lhes atribuído **null** como o valor do novo atributo.
Para removermos um atributo A de uma relação r usamos o comando **ALTER TABLE *r* DROP *A***.

2.3 Estrutura Básica de Consultas SQL

A estrutura básica de consultas SQL consiste em 3 cláusulas: **SELECT**, **FROM** e **WHERE**. Uma consulta leva como input as relações listadas na cláusula **FROM**, opera nelas como especificado nas cláusulas **WHERE** e **SELECT**, e produz uma relação como resultado.
Iremos usar a seguinte base de dados:

```
CREATE TABLE department
    (dept name VARCHAR (20),
    building VARCHAR (15),
    budget NUMERIC (12,2),
    PRIMARY KEY (dept name));
CREATE TABLE course
    (course id VARCHAR (7),
    title VARCHAR (50),
    dept name VARCHAR (20),
    credits NUMERIC (2,0),
    PRIMARY KEY (course id),
    FOREIGN KEY (dept name) REFERENCES department);
CREATE TABLE instructor
    (ID VARCHAR (5),
    name VARCHAR (20) NOT NULL,
    dept name VARCHAR (20),
    salary NUMERIC (8,2),
    PRIMARY KEY (ID),
    FOREIGN KEY (dept name) REFERENCES department);
CREATE TABLE section
    (course id VARCHAR (8),
    sec id VARCHAR (8),
    semester VARCHAR (6),
    year NUMERIC (4,0),
    building VARCHAR (15),
    room number VARCHAR (7),
    time slot id VARCHAR (4),
    PRIMARY KEY (course id, sec id, semester, year),
    FOREIGN KEY (course id) REFERENCES course);
CREATE TABLE teaches
    (ID VARCHAR (5),
    course id VARCHAR (8),
    sec id VARCHAR (8),
    semester VARCHAR (6),
    year NUMERIC (4,0),
    PRIMARY KEY (ID, course id, sec id, semester, year),
    FOREIGN KEY (course id, sec id, semester, year) REFERENCES section,
    FOREIGN KEY (ID) REFERENCES instructor);
```

2.3.1 Consultas em 1 Relação

1) Seja a seguinte consulta: "Encontra os nomes de todos os instrutores.". Os nomes dos instrutores podem ser encontrados na relação *instructor*, pelo que pomos isso na cláusula **FROM**. O nome do instrutor aparece no atributo *name*, colocando isso na cláusula **SELECT**.

```
SELECT name
FROM instructor;
```

2) Seja a seguinte consulta: "Encontra os nomes de departamento de todos os instrutores.". Dado que mais do que 1 instrutor pode pertencer ao mesmo departamento, o mesmo nome de departamento pode aparecer múltiplas vezes na relação *instructor*. Como tal, interessa-nos forçar a eliminação de duplicados na relação resultante da consulta. Para tal:

```
SELECT DISTINCT dept_name
FROM instructor;
```

3) Seja a seguinte consulta: "Como seria a relação de instrutores com um aumento de 10% no salário?". A cláusula **SELECT** permite-nos conter expressões aritméticas com os operadores $+, -, *$ e $/$. Logo:

```
SELECT ID, name, dept_name, salary * 1.1
FROM instructor;
```

4) Seja a seguinte consulta: "Encontra os names de todos os instrutores no departamento de Engenharia Informática que têm um um salário maior que 2000€.". A cláusula **WHERE** permite-nos selecionar apenas as

linhas (dados da base de dados) na relação resultante da cláusula **FROM** que satisfazem um predicado especificado. Assim:

```
01 | SELECT name
02 | FROM instructor
03 | WHERE dept_name='Engenharia InformÃ¡tica' AND salary > 2000;
```

De notar que o SQL permite o uso de conectores lógicos **and**, **or** e **not** na cláusula **WHERE**. Os operandos dos conectores lógicos podem ser expressões envolvendo operadores de comparação $<, \leq, >, \geq, =$ e $<>$, permitindo-nos comparar cadeias de caracteres e expressões aritméticas, bem como tipos especiais, como uma data.

2.3.2 Consultas em N Relações

As consultas por vezes precisam de acesso a informação de múltiplas relações.
1) Seja a seguinte consulta: "Recolha o nome de todos os instrutores, bem como o seu nome de departamento e o nome do edifício do departamento.". Analisando o esquema da relação *instructor*, percebemos que podemos obter o nome do departamento do atributo *dept_name*, mas o nome do edifício do departamento está presenta no atributo *building* da relação *department*. Para responder à consulta, cada tuplo na relação *instructor* deve ser correspondida a um tuplo da relação *department* onde os valores *dept_name* correspondam. Assim, em SQL para respondermos a esta consulta, listamos as relações necessárias na cláusula **FROM** e especificamos a condição de correspondência de valores de atributos na cláusula **WHERE**. Assim:

```
01 | SELECT name, instructor.dept_name, building
02 | FROM instructor, department
03 | WHERE instructor.dept_name = department.dept_name;
```

De notar que como o atributo *dept_name* ocorre em ambas as relações, o prefixo *instructor.dept_name* é necessário para tornar claro qual o atributo a que nos estamos a referir. Como *name* e *building* só aparecem em 1 das relações não é necessário o seu prefixo.
Podemos assim definir o papel de cada cláusula:

- **SELECT**: usada para listar os atributos desejados do resultado de uma consulta;
- **FROM**: uma lista de relações a serem acedidas na avaliação da consulta.
- **WHERE**: um predicado que envolve atributos da relação na cláusula **FROM**.

Uma consulta SQL típica tem a seguinte forma:

```
01 | SELECT A_1, A_2, ..., A_n
02 | FROM r_1, r_2, ..., r_m
03 | WHERE P;
```

Cada A_i representa um atributo, cada r_i representa uma relação e P é um predicado - se P for omitido, este é considerado **true**.
A clásula **FROM** define um **produto Cartesiano** entre as relações listadas na cláusula. Pode ser entendido pelo seguinte processo iterativo que gera tuplo para a relação resultante da clásula **FROM**:

```
for each tuple t_1 in relation r_1:
    for each tuple t_2 in relation r_2:
        ...
            for each tuple t_m in relation r_m:
                concatenate t_1, t_2, ..., t_m into a single tuple t
                add t into result relation
```

Se quisermos fazer o produto cartesiano entre as relações *instructor* e *teaches*, teremos o seguinte esquema relacional:

```
(instructor.ID, instructor.name, instructor.dept_name, instructor.salary,
teaches.ID, teaches.course_id, teaches.sec_id, teaches.semester, teaches.year)
```

Pelo que simplificando tendo em conta os atributos que só aparecem em 1 das relações:

`(instructor.ID, name, dept_name, salary, teaches.ID, course_id, sec_id, semester, year)`

O produto Cartesiano resultante combina tuplos que não têm qualquer tipo de relação entre eles - o resultado pode ser uma relação extremamente grande, e raramente faz sentido criar tal produto Cartesiano.

instructor.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2017
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2018
12121	Wu	Finance	90000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2018
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
15151	Mozart	Music	40000	10101	CS-101	1	Fall	2017
15151	Mozart	Music	40000	10101	CS-315	1	Spring	2018
15151	Mozart	Music	40000	10101	CS-347	1	Fall	2017
15151	Mozart	Music	40000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
15151	Mozart	Music	40000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
22222	Einstein	Physics	95000	10101	CS-101	1	Fall	2017
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2018
22222	Einstein	Physics	95000	10101	CS-347	1	Fall	2017
22222	Einstein	Physics	95000	12121	FIN-201	1	Spring	2018
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2018
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Figura 2.1: Produto cartesiano da relação *instructor* com a relação *teaches*

A cláusula **WHERE** é usada para restringir as combinações criadas pelo produto Cartesiano ao essencial. **2)** Seja a seguinte consulta: "Lista todos os instrutores bem como o curso que estes lecionam.". Temos:

```
01 |      SELECT name, course_id
02 |      FROM instructor, teaches
03 |      WHERE instructor.ID = teaches.ID;
```

3) Seja a seguinte consulta: "Lista todos os cursos que os instrutores do departamento de Engenharia Informática lecionam.". Temos:

```
01 |      SELECT name, course_id
02 |      FROM instructor, teaches
03 |      WHERE instructor.ID = teaches.ID AND dept_name='Engenharia Informatica';
```

Podemos assim generalizar o fluxo de uma consulta SQL como:

1 Gerar um produto Cartesiano nas relações listadas na cláusula **FROM**;

2 Aplicar os predicados especificados na cláusula **WHERE** como resultado do passo 1;

3 Para cada tuplo no resultado do passo 2, dar output dos atributos espeficados na cláusula **SELECT**.

2.4 Operações Básicas Adicionais

2.4.1 Operação de Renomeação

Os nomes dos atributos no resultado de uma consulta derivam dos nomes dos atributos das relações especificadas na cláusula **FROM**. Porém, nem sempre podemos derivar nomes desta maneira: duas relações na cláusula

FROM podem ter atributos com o mesmo nome; se usarmos uma expressão aritmética na cláusula **SELECT** o atributo resultante não tem um nome; se um nome de um atributo pode ser derivado diretamente da relação base (por exemplo um nome de atributo presente em apenas 1 relação), podemos querer mudar o nome do atributo no resultado.
A cláusula **AS** pode aparecer quer nas cláusulas **SELECT**, quer nas **FROM**.
1) Se quisermos substituir um nome de atributo por algo mais específico:

```
01 |  SELECT name AS instructor_name, course_id
02 |  FROM instructor, teaches
03 |  WHERE instructor.ID = teaches.ID;
```

2) Seja a seguinte consulta: "Para todos os instrutores na universidade que ensinam um curso, encontra os nomes e o ID do curso que lecionam.". Temos:

```
01 |  SELECT T.name, S.course_id
02 |  FROM instructor AS T, teaches AS S
03 |  WHERE T.ID = S.ID;
```

3) Seja a seguinte consulta: "Encontra o nome de todos os instrutores cujo salário é maior que pelo menos 1 instrutor no departamento de Biologia.". Temos:

```
01 |  SELECT DISTINCT T.name,
02 |  FROM instructor AS T, instructor AS S
03 |  WHERE T.salary > S.salary AND S.dept_name='Biologia';
```

No exemplo acima, **T** e **S** são consideradas cópias da relação *instructor*, mas mais precisamente, são declarados como ***alias***, i.e., nomes alternativas à relação *instructor* - **T** e **S** são referidos como **nomes de correlação** em SQL *standard*, mas também são referidos como ***table alias***, ***correlation variable*** ou ***tuple variable***.

2.4.2 Operações em Cadeias de Caracteres

O SQL especifica cadeias de caracteres com aspas singulares, i.e., 'Exemplo'. Para especificarmos uma aspa singular dentro duma cadeia de caracteres devemos usar 'It"s right'.
O SQL *standard* especifica que as operações de igualdade em cadeias de caracteres são sensíveis a maiúsculas/minúsculas.
Permite também uma variedade de funções em cadeias de caracteres, tal como **concatenação** (||), extrair *sub-strings*, encontrar o **comprimento** de cadeias de caracteres, converter cadeias de caracteres para tudo **maiúsculo** (**upper**(s) onde s é uma cadeia de caracteres), **minúsculo** (**lower**(s)), remover espaços no fim da cadeia de caracteres (**trim**(s)), e por aí em diante.
O SQL fornece ainda correspondência de padrões em cadeias de caracteres usando o operador **LIKE**. Descrevemos padrões usando 2 caracteres especiais:

- **Percentagem (%)**: corresponde a qualquer *sub-string*;
- ***Underscore*** **(_)**: corresponde a qualquer caracter.

Alguns exemplos que ilustram a correspondência de padrões em SQL:

- 'Intro%': corresponde a qualquer cadeia de caracteres que começo com "Intro".
- '%Comp%': corresponde a qualquer cadeia de caracteres que contém "Comp"como *sub-string*, como por exemplo "Intro. to Computer Science"e "Computational Biology".
- '___': corresponde a uma cadeia de caracteres de exatamente 3 caracteres.
- '___%': corresponde a uma cadeia de caracteres de pelo menos 3 caracteres.

1) Seja a seguinte consulta: "Encontra os nomes de todos os departamentos cujos edifícios têm o nome 'Watson'".
Temos:

```
01 |  SELECT dept_name
02 |  FROM department
03 |  WHERE building LIKE '%Watson%';
```

O SQL permite o uso de um caracter *escape* que antecede qualquer caracter especial de padrões para indicar que o caracter especial de padrões está a ser tratado como um caracter normal - para tal usamos a palavra-chave **ESCAPE**:

- **LIKE** 'ab\%cd%' ESCAPE '\': corresponde a todas as cadeias de caracteres que começam com "ab%cd".
- **LIKE** 'ab\\cd%' ESCAPE '\': corresponde a todas as cadeias de caracteres que começam com "ab\cd".

2.4.3 Especificação de Atributo na cláusula SELECT

O símbolo do asterisco (*) pode ser usado na cláusula **SELECT** para denotar todos os atributos. Podemos especificar então uma consulta com 2 relações em que queiramos todos os atributos de apenas 1 das relações da seguinte maneira:

```
01 |  SELECT instructor.*
02 |  FROM instructor, teaches
03 |  WHERE instructor.ID = teaches.ID;
```

2.4.4 Ordenação da Disposição dos Tuplos

O SQL oferece ao utilizador algum controlo sobre a ordem na qual os tuplos numa relação são dispostos - a cláusula **ORDER BY** causa os tuplos no resultado da query a seguirem uma dada relação de ordenação.
1) Seja a seguinte consulta: "Liste todos os instrutores que trabalham no Departamento de Física por ordem alfabética.". Temos:

```
01 |  SELECT name
02 |  FROM instructor
03 |  WHERE dept_name = 'FÃsica'
04 |  ORDER BY name;
```

Se tivermos vários nomes de atributos na cláusula **ORDER BY** este segue o primeiro como principal critério de ordenação e os subsequentes como critério de desempate.
2) Seja a seguinte consulta: "Liste a relação *instructor* por ordem descendente de salário, e em caso de empate, por ordem alfabética.". Temos:

```
01 |  SELECT *
02 |  FROM instructor
03 |  ORDER BY salary DESC, name ASC;
```

2.4.5 Predicados da cláusula WHERE

O SQL inclui um operador **BETWEEN** para simplificar enquadramentos de valores.
1) Seja a seguinte consulta: "Liste todos os instrutores cujos salários estão entre 2000€ e 3000€.". Temos:

```
01 |  SELECT name
02 |  FROM instructor
03 |  WHERE salary BETWEEN 2000 AND 3000;
```

Similarmente, podemos usar o operador de comparação **NOT BETWEEN**.
O SQL permite-nos ainda usar a notação $(v_1, v_2, ..., v_n)$ para denotar um tuplo de aridade n contendo os valores $v_1, v_2, ..., v_n$ - esta notação é designada ***row constructor***. Por exemplo, $(a_1, a_2) \leq (b_1, b_2)$ só é verdade sse $a_1 \leq b_1$ e $a_2 \leq b_2$.
Podemos então reescrever este query

```
01 |  SELECT name, course_id
02 |  FROM instructor, teaches
03 |  WHERE instructor.ID = teaches.ID AND dept_name = 'Biologia';
```

usando a notação do ***row constructor***:

```
SELECT name, course_id
FROM instructor, teaches
WHERE (instructor.ID, dept_name) = (teaches.ID, 'Biologia');
```

2.4.6 Operações de Conjuntos

As operações SQL **union**, **intersect** e **except** operam em relações e correspondem às operações matemáticas de conjuntos $\cup, \cap$ e $-$.

Operador de União

1) Seja a seguinte consulta: "Selecione todos os ID's de cursos lecionados no 2º semestre do ano de 2017 e 1ºsemestre do ano de 2018.". Podemos decompor isto:

```
-- O conjunto de cursos lecionados no 2 semestre de 2017
SELECT course_id
FROM section
WHERE semester=2 AND year=2017;

-- O conjunto de cursos lecionados no 1 semestre de 2018
SELECT course_id
FROM section
WHERE semester=1 AND year=2018;

-- A uniao destes da-nos o resultado da consulta (sem duplicados)
(SELECT course_id
FROM section
WHERE semester=2 AND year=2017)
UNION
(SELECT course_id
FROM section
WHERE semester=1 AND year=2018);

-- A uniao destes da-nos o resultado da consulta (com duplicados)
(SELECT course_id
FROM section
WHERE semester=2 AND year=2017)
UNION ALL
(SELECT course_id
FROM section
WHERE semester=1 AND year=2018);
```

De notar que o operador **UNION** elimina automaticamente os duplicados, pelo que se os quisermos manter deve suceder ao operador a palavra-chave **ALL**.

Operador de Interseção e de Exceção

Os operadores de interseção e exceção são semanticamente iguais ao da união, pelo que o código acima pode ser reutilizado substituindo apenas a palavra-chave **UNION** por **INTERSECT** ou **EXCEPT** (o **ALL** tem o mesmo propósito para todos os operadores), mediante o propósito da consulta.

2.5 Valores NULL

Os **valores NULL** apresentam problemas nas operações relacionais, incluindo operações aritméticas, de comparação e de conjuntos.
O resultado de uma expressão aritmética em que um dos valores de *input* é NULL, é também NULL (exemplo: $r.A + 5$ onde $r.A$ é null para um tuplo em particular - a expressão resultante é também null para esse tuplo).
O resultado de uma operação de comparação é **unknown** sempre que um dos valores de *input* é null.
Os predicados nas cláusulas **WHERE** envolvem operações Booleanas como **AND**, **OR** e **NOT**. Estes estão prontos para lidar com o valor **unknown** da seguinte forma:

- **AND**: O resultado de *true* **AND** *unknown* é *unknown* e *false* **AND** *unknown* é *false*, enquanto que *unknown* **AND** *unknown* é *unknown*.
- **OR**: O resultado de *true* **OR** *unknown* é *true* e *false* **OR** *unknown* é *unknown*, enquanto que *unknown* **OR** *unknown* é *unknown*.
- **NOT**: O resultado de **NOT** *unknown* é *unknown*.

Se o predicado da cláusula **WHERE** avalia a **false** ou **unknown** para um tuploe, este não é adicionado ao resultado.
O SQL usa a palavra-especial **NULL** num predicado para testar para um valor null.
1) Seja a seguinte consulta: "Liste todos os instrutores que não têm um salário especificado.". Temos:

```
01 |  SELECT name
02 |  FROM instructor
03 |  WHERE salary IS NULL;
```

De notar que o salário não pode ser comparado em igualdade a um valor nulo, isto é, não poderíamos ter $salary = NULL$, e como tal, usamos a palavra-chave **IS**.

2.6 Funções de Agregação

As **funções de agregação** são funções que tomam uma coleção de valores como *input* e retornam um único valor. O SQL oferece 5 funções de agregação:

- **Média**: AVG (requer que a coleção seja unicamente composta de números)
- **Mínimo**: MIN
- **Máximo**: MAX
- **Total**: SUM (requer que a coleção seja unicamente composta de números)
- **Contagem**: $COUNT$

2.6.1 Agregação Básica

1) Seja a seguinte consulta: "Encontra o salário médio dos instrutores no departamento de Engenharia Informática.". Temos:

```
01 |  SELECT AVG(salary)
02 |  FROM instructor
03 |  WHERE dept_name='Engenharia Informatica';
04 |
05 |  -- Porem, queremos dar um nome com significado ao resultado do SELECT
06 |  SELECT AVG(salary) AS avg_salary
07 |  FROM instructor
08 |  WHERE dept_name='Engenharia Informatica';
```

2) Seja a seguinte consulta: "Encontra o número total de instrutores que lecionaram um curso no 1º semestre de 2018.". Temos:

```
01 |  -- O DISTINCT aqui permite que um instrutor tenha lecionado mais que 1 curso, mas ser
       contado apenas 1 vez
02 |  SELECT COUNT(DISTINCT id)
03 |  FROM teaches
04 |  WHERE semester = 1 AND year = 2018;
```

3) Seja a seguinte consulta: "Quantas entradas estão na relação *course*?". Temos:

```
01 |  SELECT COUNT(*)
02 |  FROM course;
```

De notar que o SQL não permite o use da palavra-chave **DISTINCT** com a função de agregação **COUNT(*)**.

2.6.2 Agregação com Agrupamento

Existem circunstâncias em que queremos aplicar a função de agregação não só a um único conjunto de tuplo, mas também a grupos de conjuntos de tuplos - especificamos isto em SQL usando a cláusula **GROUP BY**.
Os atributos dados na cláusula **GROUP BY** são usados para formar grupos. Tuplos com o mesmo valor em todos os atributos na cláusula **GROUP BY** são postos num grupo. **1)** Seja a seguinte consulta: "Encontra o salário médio para cada departamento.". Temos:

```
01 |  SELECT dept_name, AVG(salary) AS avg_salary
02 |  FROM instructor
03 |  GROUP by dept_name;
```

2) Seja a seguinte consulta: "Encontra o número de instrutores em cada departamento que lecionaram um curso no 1º semestre de 2018.". Temos:

```
01 |  SELECT dept_name, COUNT(DISTINCT id) AS instr_count
02 |  FROM instructor, teaches
03 |  WHERE (instructor.ID, semester, year) = (teaches.ID, 1, 2018)
04 |  GROUP BY dept_name;
```

É importante ao usar agrupamento em consultas SQL assegurar que os únicos atributos que aparecem no cláusula **SELECT**, e que não são as que estão a ser agregadas, estão presentes na cláusula **GROUP BY**.

2.6.3 Cláusula HAVING

Quando queremos aplicar condições que se aplicam a **grupos** e não tuplos, usamos a cláusula **HAVING**. Recordemos que os grupos são formados através da cláusula **GROUP BY**.
1) Seja a seguinte consulta: "Encontra o salário médio dos instrutores nos departamentos cujo salário médio é maior que 2500€.". Temos:

```
01 |  SELECT dept_name, AVG(salary) AS avg_salary
02 |  FROM instructor
03 |  GROUP BY dept_name
04 |  HAVING AVG(salary) > 42000;
```

É importantíssimo notar que o SQL aplica os predicados na cláusula **HAVING após os grupos se formarem**, pelo que funções de agregação podem ser usadas na cláusula **HAVING**.
2) Seja a seguinte consulta: "Para cada curso oferecido em 2017, encontra a média de créditos totais (tot_cred) de todos os estudantes inscritos na secção, se a secção tem pelo menos 2 estudantes.". Temos:

```
01 |  SELECT course_id, semester, year, sec_id, AVG(tot_cred)
02 |  FROM student, takes
03 |  WHERE (student.ID, year) = (takes.ID, 2017)
04 |  GROUP BY course_id, semester, year, sec_id
05 |  HAVING COUNT(ID) >= 2;
```

2.6.4 Agregação com Valores NULL e Booleanos

As funções de agregação tratam o valor null do seguinte modo: todas as funções de agregação exceto o **COUNT (*)** ignoram valores null para a coleção de input. Como resultado de ignorar null values, a coleção de valores pode ser vazia. O **COUNT** de uma coleção vazia é definido como 0, e todas as outras operações de agregação retorna o valor de null quando aplicado numa coleção vazia.
O tipo de dados Booleano pode assumir valores **true**, **false** e **unknown**. As funções de agregação **SOME** e **ALL** podem ser aplicadas a uma coleção de valores Booleanos, e computar a disjunção (**OR**) e a conjunção (**AND**), respetivamente, dos valores.

2.7 Sub-Consultas Aninhadas

O SQL fornece um mecanismo de aninhamento de sub-consultas.

Definição 6 (Sub-Consulta)**.** Uma sub-consulta é uma expressão **SELECT-FROM-WHERE** que está aninhada noutra consulta.

As sub-consultas são geralmente usadas para executar testes em pertença de conjuntos, comparações entre conjuntos e determinação de cardinalidade de conjuntos.

2.7.1 Pertença de Conjuntos

O SQL permite testar pertença de tuplos numa relação. O conector de palavra-chave **IN** testa por pertença de conjuntos, onde o conjunto é uma coleção de valores produzida pela cláusula **SELECT**. O conector **NOT IN** testa a abstenção de pertença.
1) Seja de novo a seguinte consulta: "Selecione todos os ID's de cursos lecionados no 2º semestre do ano de 2017 e 1ºsemestre do ano de 2018.". Podemos reescrever a consulta do seguinte modo:

```
01 |  SELECT DISTINCT course_id
02 |  FROM section
03 |  WHERE semester=2 AND year=2017 AND
04 |      course_id IN (SELECT course_id
05 |                    FROM section
06 |                    WHERE semester=1 AND year=2018);
```

De notar que os operadores **IN** e **NOT IN** podem também ser usados em conjuntos enumerados. Por exemplo,
2) Seja a seguinte consulta: "Encontra o nome de todos os instrutores cujo nome não é "Mozart"nem "Einstein . Temos:

```
01 |  SELECT DISTINCT name
02 |  FROM instructor
03 |  WHERE name NOT IN ('Mozart', 'Einstein');
```

2.7.2 Comparação de Conjuntos

1) Seja de novo a seguinte consulta: "Encontra o nome de todos os instrutores cujo salário é maior que pelo menos 1 instrutor no departamento de Biologia.". Temos:

```
01 |  SELECT name
02 |  FROM instructor
03 |  WHERE salary > SOME (SELECT salary
04 |                       FROM instructor
05 |                       WHERE dept_name='Biologia');
```

O operador $> SOME$ na cláusula **WHERE** no **SELECT** exterior é verdadeiro se o valor de *salary* do tuplo é maior que pelo menos um membro do conjunto de todos os valores de salários de instrutores em Biologia.
2) Seja a seguinte consulta: "Encontra o nome de todos os instrutores que têm um salário maior do que todos os instrutores no departamento de Biologia.". Temos:

```
01 |  SELECT name
02 |  FROM instructor
03 |  WHERE salary > ALL (SELECT salary
04 |                      FROM instructor
05 |                      WHERE dept_name='Biologia');
```

3) Seja a seguinte consulta: "Encontra os departamentos que têm o maior salário médio.". Temos:

```
01 |  SELECT dept_name
02 |  FROM instructor
03 |  GROUP BY dept_name
04 |  HAVING AVG(salary) >= ALL (SELECT AVG(salary)
```

```
05 |                              FROM instructor
06 |                              GROUP BY dept_name);
```

2.7.3 Teste para Relações Vazias

O SQL inclui um elemento para testar se uma sub-consulta tem algum tuplo no seu resultado - palavra-chave **EXISTS**. É um booleano que retorna o valor **true** se a sub-consulta de argumento não é vazia.
1) Seja de novo a seguinte consulta: "Selecione todos os ID's de cursos lecionados no 2º semestre do ano de 2017 e 1ºsemestre do ano de 2018.". Temos agora:

```
01 |  SELECT course_id
02 |  FROM section AS S
03 |  WHERE semester=2 AND year=2017 AND
04 |      EXISTS (SELECT *
05 |              FROM section AS T
06 |              WHERE semester=1 AND year=2018 and S.course_id = T.course_id);
```

Isto é a primeira vez que uma consulta ilustra o funcionamento de um **nome de correlação** de uma consulta externa (**S**) é usado numa sub-consulta na cláusula **WHERE**. Uma sub-consulta que usa o nome de correlação de uma consulta externa é chamada uma **sub-consulta correlacionada**.
2) Seja a seguinte consulta: "Encontra todos os alunos que tiraram cursos oferecidos pelo departamento de Biologia.". Temos:

```
01 |  SELECT S.ID, S.name
02 |  FROM student AS S
03 |  WHERE NOT EXISTS ((SELECT course_id
04 |                     FROM course
05 |                     WHERE dept_name='Biologia'
06 |                     EXCEPT
07 |                     (SELECT T.course_id
08 |                     from takes AS T
09 |                     WHERE S.ID=T.ID));
10 |  -- Retorna o conjunto de cursos oferecidos pelo departamento de Biologia
11 |  SELECT course_id
12 |  FROM course
13 |  WHERE dept_name='Biologia'
14 |
15 |  -- Retorna o conjunto de cursos que o estudante S.ID jÃ¡ tirou
16 |  SELECT T.course_id
17 |  FROM takes AS T
18 |  WHERE S.ID=T.ID
```

2.7.4 Teste para Ausência de Tuplos Duplicados

O SQL inclui uma função Booleana para testar se uma sub-consulta tem tuplos duplicados no seu resultado - o construtor **UNIQUE** retorna o valor **true** se a sub-consulta não contém tuplos duplicados.
1) Seja a seguinte consulta: "Encontra todos os cursos que foram oferecidos no máximo 1 vez em 2017.". Temos:

```
01 |  SELECT T.course_id
02 |  FROM course AS T
03 |  WHERE UNIQUE (SELECT R.course_id
04 |                FROM section AS R
05 |                where T.course_id = R.course_id AND R.year=2017);
06 |
07 |  -- Equivalentemente podemos usar o COUNT
08 |  SELECT T.course_id
09 |  FROM course AS T
10 |  WHERE 1 >= (SELECT COUNT(R.course_id)
11 |                FROM section AS R
12 |                where T.course_id = R.course_id AND R.year=2017);
```

2.7.5 Sub-Consultas na cláusula FROM

O SQL permite uma sub-consulta ser usada como expressão na cláusula **FROM**. O conceito-chave aplicado aqui é que a expressão **SELECT-FROM-WHERE** retorna uma relação como resultado, e, como tal, pode ser inserido noutro **SELECT-FROM-WHERE** em qualquer lado que uma relação possa aparecer.
1) Seja a seguinte consulta: "Encontra o salário médio dos instrutores cujos departamentos têm um salário médio maior que 2500€.". Temos:

```
01 | SELECT dept_name, avg_salary
02 | FROM (SELECT dept_name, AVG(salary) as avg_salary
03 |       FROM instructor
04 |       GROUP BY dept_name) AS dept_avg (dept_name, avg_salary)
05 | WHERE avg_salary > 42000;
```

De notar que no *PostgreSQL* (SGBD da cadeira) é necessário que cada relação de sub-consulta numa cláusula **FROM** lhe seja atribuído um nome, mesmo que nunca seja referenciado - neste caso usámos o identificador **dept_avg**.
2) Seja a seguinte consulta: "Encontra o máximo de salário que cada departamento com todos os seus instrutores tem.". Temos:

```
01 | SELECT MAX(tot_salary)
02 | FROM (SELECT dept_name, SUM(salary)
03 |       FROM instructor
04 |       GROUP BY dept_name) AS dept_avg (dept_name, tot_salary);
```

2.7.6 Claúsula WITH

A cláusula **WITH** fornece uma maneira de definir temporariamente uma relação cuja definição só está disponível à consulta na qual a cláusula **WITH** ocorre.
1) Seja a seguinte consulta: "Encontra os departamentos com maior salário.". Temos:

```
01 | WITH max_budget(value) AS
02 |     (SELECT MAX(budget)
03 |      FROM departament)
04 | SELECT budget
05 | FROM department, max_budget
06 | WHERE department.budget = max_budget.value;
```

2) Seja a seguinte consulta: "Encontra os departamentos cujo salário total é maior do que a média de salário total em todos os departamentos.". Temos:

```
01 | WITH dept_total(dept_name, value) AS
02 |     (SELECT dept_name, SUM(salary)
03 |      FROM instructor
04 |      GROUP BY dept_name),
05 |      dept_total_avg(value) AS
06 |     (SELECT AVG(value)
07 |      FROM dept_total)
08 | SELECT dept_name
09 | FROM dept_total, dept_total_avg
10 | WHERE dept_total.value > dept_total_avg.value;
```

2.7.7 Sub-Consultas Escalares

O SQL permite que sub-consultas ocorram sempre que uma expressão que retorna um valor é permitida, desde que a sub-consulta retorna apenas um tuplo contendo um único atributo - tais sub-consultas são designadas **sub-consultas escalares**.
1) Seja a seguinte consulta: "Lista todos os departamentos bem como o número de instrutores em cada departamento.". Temos:

```
SELECT dept_name,
       (SELECT COUNT(*)
        FROM instructor
        WHERE department.dept_name=instructor.dept_name) AS num_instructors
FROM department;
```

2.8 Modificação da Base de Dados

Vejamos agora como adicionar, remover ou mudar informação com o SQL.

2.8.1 Remoção

Um pedido **DELETE** é expresso da mesma maneira que uma consulta - só podemos **apagar tuplos**; não podemos apagar valores de certos atributos apenas. SQL expressa uma remoção por:

```
DELETE FROM r
WHERE P;
```

onde **P** é um predicado e **r** representa uma relação.
O **DELETE** procura primeiro todos os tuplos t em **r** tal que ***P(t)*** seja verdadeiro, e depois remove-os de **r**. Se o predicado **P** for omitido, remove todos os tuplos da relação **r**.
1) Seja a seguinte operação: "Apague todos os tuplos na relação *instructor* cujo edifício do respetivo departamento contém Watson no nome.". Temos:

```
DELETE FROM instructor
WHERE dept_name IN (SELECT dept_name
                    FROM department
                    WHERE building LIKE '%Watson%');
```

2.8.2 Inserção

Para inserir dados numa relação, ou especificamos o tuplo a ser inserido ou escrevemos uma consulta cujo resultado é um conjunto de tuplos a serem inseridos.
1) Seja a seguinte operação: "Insira um novo curso de ID EI-437 no departamento de Engenharia Informática com título "Sistemas de Bases de Dados"e 4 horas de crédito. Escrevemos:

```
INSERT INTO course
       VALUES ('EI-437', 'Sistemas de Bases de Dados', 'Engenharia Informatica', 4);
```

2) Seja a seguinte operação: "Faça cada estudante no departamento de Música que já tenha ganho 144 horas de crédito um instrutor no departamento de Música com um salário de 800€.". Temos:

```
INSERT INTO instructor
       SELECT ID, name, dept_name, 18000
       FROM student
       WHERE dept_name='Musica' AND tot_cred > 144;
```

2.8.3 Atualizações

Em certas situações, podemos querer mudar o valor de um tuplo sem mudar **todos** os valores nesse mesmo. Para tal, usamos o **STATEMENT**.
1) Seja a seguinte operação: "Faça cada instrutor ter um aumento de 5% no seu salário.". Temos:

```
UPDATE instructor
SET salary = salary * 1.05

-- Um tweak seria atualizar o salario apenas para quem recebe menos de 1000 pau
```

```
05 | UPDATE instructor
06 | SET salary = salary * 1.05
07 | WHERE salary < 1000;
08 |
09 | -- Outro tweak seria atualizar o salario apenas a quem esta abaixo da media
10 | UPDATE instructor
11 | SET salary = salary * 1.05
12 | WHERE salary < (SELECT AVG(salary)
13 |                 FROM instructor);
14 |
15 | -- Por fim um tweak cuja ORDEM IMPORTA seria
16 | UPDATE instructor
17 | SET salary = salary * 1.03
18 | WHERE salary > 1000;
19 |
20 | UPDATE instructor
21 | SET salary = salary * 1.05
22 | WHERE salary <= 1000;
```

Como vimos, no último *tweak* a ordem era relevante. Para facilitar a vida do programador, o SQL fornece um construtor **CASE** para executar ambas as atualizações num único **UPDATE**:

```
01 | UPDATE instructor
02 | SET salary = CASE
03 |                 WHEN salary <= 1000 THEN salary * 1.05
04 |                 ELSE salary * 1.03
05 |              END
06 |
07 | -- A forma geral do CASE eh dada por
08 |              CASE
09 |                 WHEN pred_1 THEN res_1
10 |                 WHEN pred_2 THEN res_2
11 |                 ...
12 |                 WHEN pred_n THEN res_n
13 |                 ELSE res_0
14 |              END
```

2.9 Expressões JOIN

Até agora, usámos o produto Cartesiano para combinar informação de múltiplas relações. O operador **JOIN** permite escrever consultas com múltiplas relações de um modo mais natural.

2.9.1 Natural JOIN

A operação de **NATURAL JOIN** opera em 2 relações e produz uma relação como resultado. Considera apenas os pares de tuplos com o mesmo valor nos atributos que aparecem nos esquemas de ambos.
Assim,
1) Seja a seguinte consulta: "Para todos os estudantes na universidade que tiraram um curso, descobre o nome e o ID do curso que tiraram.". Temos:

```
01 | -- Antes do JOIN
02 | SELECT name, course_id
03 | FROM student, takes
04 | WHERE student.ID = takes.ID
05 |
06 | -- Com Natural JOIN
07 | SELECT name, course_id
08 | FROM student NATURAL JOIN takes
09 |
10 | -- AMBAS AS QUERIES GERAM O MESMO RESULTADO
```

De notar que o que realmente está a acontecer ao fazer **NATURAL JOIN** é que se estão a considerar apenas os pares de tuplos onde quer o tuplo de *student*, quer o tuplo de *takes* têm o mesmo valor no atributo comum, **ID**.

O resultado de uma operação de **NATURAL JOIN** é uma relação. Uma cláusula **FROM** de uma consulta SQL pode ter múltiplas relações combinadas usando o **NATURAL JOIN**. A consulta SQL na sua forma geral usando **NATURAL JOIN**'s é dada por:

```
01 | SELECT A_1, A_2, ..., A_n
02 | FROM E_1, E_2, ..., E_m
03 | WHERE P;
```

Por sua vez, cada E_i pode ser uma única relação ou uma expressão envolvendo **NATURAL JOIN**'s.
1) Seja a seguinte consulta: "Lista os nomes dos estudantes bem como o título dos cursos que já tiraram.". Temos:

```
01 | -- Exemplificacao da formula geral (primeiro faz o NATURAL JOIN, so depois o produto
       cartesiano)
02 | SELECT name, title
03 | FROM student NATURAL JOIN takes, course
04 | WHERE takes.course_id = course.course_id;
```

De notar que não podemos fazer um duplo **NATURAL JOIN**, pois necessitaríamos que os valores dos atributos *dept_name* e *course_id* fossem iguais, quando só queremos juntar o *course_id*.
Para tal, o SQL fornece a operação **JOIN ... USING** que leva no **USING** uma lista de nomes de atributos a serem especificados. Ou seja, se tivéssemos

```
01 | r_1 JOIN r_2 USING (A_1, A_2)
```

seria semelhante a

```
01 | r_1 NATURAL JOIN r_2
```

porém um par de tuplos no primeiro caso corresponde se $t_1.A_1 = t_2.A_1$ e $t_1.A_2 = t_2.A_2$, e mesmo que se tivessem ambos um terceiro atributo A_3, não seria necessário $t_1.A_3 = t_2.A_3$, enquanto que no segundo caso já seria.

2.9.2 Condições JOIN

O SQL suporta outra forma de **JOIN**, no qual uma condição arbitrária pode ser especificada com a palavra-chave **ON**. Esta palavra-chave recebe um predicado geral sobre as relações a serem **JOIN**ed e aparece no fim da expressão **JOIN**.
0) Seja a seguinte consulta:

```
01 | SELECT *
02 | FROM student JOIN takes ON student.ID = takes.ID;
03 |
04 | -- Equivalente a
05 | SELECT *
06 | FROM student, takes
07 | WHERE student.ID = takes.ID;
```

De notar que o que difere isto de um **NATURAL JOIN** é que a relação resultante tem o atributo ID listado 2 vezes, 1 para *student* e outro para *takes*, apesar dos seus valores terem de ser o mesmo.

```
01 | -- Alternativa ao problema supramencionado
02 | SELECT student.ID as ID, name, dept_name, tot_cred,
03 |        course_id, sec_id, semester, year, grade
04 | FROM student JOIN takes ON student.ID = takes.ID;
```

2.9.3 Outer JOIN

A operação **OUTER JOIN** funciona de forma similar às que já vimos, mas preserva os tuplos que seriam perdidos num **JOIN** ao criar tuplos no resultado que contêm **valores null**. É importante referir que quer o **NATURAL JOIN**, quer o **JOIN ... ON** não conseguiriam corresponder um valor null a outro não null.
Existem 3 formas de **OUTER JOIN** (consideremos **A <keyword> OUTER JOIN B**):

- **LEFT OUTER JOIN**: preserva os tuplos apenas da relação A;
- **RIGHT OUTER JOIN**: preserva os tuplos apenas da relação B;
- **FULL OUTER JOIN**: preserva os tuplos das relações A e B;

Em contraste, as operações **JOIN . . . USING**, **NATURAL JOIN** e **JOIN . . . ON P** são chamadas operações **INNER JOIN**.
A operação de **LEFT OUTER JOIN** opera do seguinte modo:

```
res = A NATURAL JOIN B
for each tuple r in A that doesn't match with B:
    r->derived_from_A = r->derived_from_A
    (r->derived_from_B \ r->derived_from_A) = null
    // considera-se r = r->derived_from_A U r->derived_from_B
    res += {r}
```

Se considerarmos um estudante que nunca tenha tirado um curso, podemos agora listar todos os estudantes e os seus cursos (mesmo que ainda não tenham tirado um), do seguinte modo:

```
01 |  SELECT *
02 |  FROM student NATURAL LEFT OUTER JOIN takes;
```

1) Seja a seguinte consulta: "Encontra todos os alunos que ainda não tiraram um curso.". Temos:

```
01 |  -- Usando LEFT OUTER JOIN
02 |  SELECT ID
03 |  FROM student NATURAL LEFT OUTER JOIN takes
04 |  WHERE course_id IS NULL;
05 |
06 |  -- Usando RIGHT OUTER JOIN (simetrico)
07 |  SELECT ID
08 |  FROM takes NATURAL RIGHT OUTER JOIN student
09 |  WHERE course_id IS NULL;
```

O **FULL OUTER JOIN** é uma combinação de **LEFT OUTER JOIN** e **RIGHT OUTER JOIN**. Depois de computar o resultado do **INNER JOIN**, extende com **nulls** os tuplos do lado esquerdo da relação que não corresponderam com nenhum do lado direito da relação, e vice-versa. Por outras palavras, **FULL OUTER JOIN** = **LEFT OUTER JOIN** ∪ **RIGHT OUTER JOIN**.
2) Seja a seguinte consulta: "Exiba uma lista de todos os estudantes no departamento de Engenharia Informática, bem como as secções de curso, se alguma, que ocorreram no 2º semestre de 2017; todas as secções de curso do 2º semestre de 2017 devem ser dispostas, mesmo que nenhum estudante de Engenharia Informática tenha tirado a secção do curso.". Temos:

```
01 |  SELECT *
02 |  FROM (SELECT *
03 |        FROM STUDENT
04 |        WHERE dept_name='Engenharia Informatica')
05 |        NATURAL FULL OUTER JOIN
06 |        (SELECT *
07 |         FROM TAKES
08 |         WHERE semester=2 AND year=2017);
```

A cláusula **ON** pode ser usada com **OUTER JOIN**'s. É importante notar que este difere do modo como a cláusula **WHERE** opera, ao contrário dos **INNER JOIN**. Ou seja,

```
01 |  -- LEFT OUTER JOIN (ON)
02 |  SELECT *
03 |  FROM student LEFT OUTER JOIN takes ON student.ID = takes.ID;
04 |
05 |  -- LEFT OUTER JOIN (WHERE)
06 |  SELECT *
07 |  FROM student LEFT OUTER JOIN takes ON true
08 |  WHERE student.ID = takes.ID
```

O caso do **ON** tem um tuplo com estudantes que ainda não tenham tirado nenhum curso. No caso do **WHERE**, o *takes ON true* faz com que o **LEFT OUTER JOIN** se comporte como um produto Cartesiano das 2 relações. Seja um estudante que ainda não tenha tirado um curso com ID = 69. A cláusula **WHERE** não irá encontrar nenhuma correspondência entre *student*.ID e *takes*.ID, pois não existe nenhum tuplo em *takes* com ID = 69.

2.9.4 Tipos e Condições de JOIN

Para distinguir **JOIN**'s normais de **OUTER JOIN**'s, os **JOIN**'s normais são designados **INNER JOIN**'s em SQL.
Uma cláusula de **JOIN** pode ser usada para especificar **INNER JOIN** em vez de **OUTER JOIN** para espeficar que se quer um **JOIN** normal. Contudo, a palavra-chave **INNER** é opcional, pois por defeito, a cláusula **JOIN** efetua um **INNER JOIN** ou **JOIN** normal, ou seja,

```
01 |  -- JOIN
02 |  SELECT *
03 |  FROM student JOIN takes USING (id);
04 |
05 |  -- INNER JOIN
06 |  SELECT *
07 |  FROM student INNER JOIN takes USING (id);
```

estas 2 consultas são absolutamente equivalentes.
Eis uma lista que mostra que os vários tipos de **JOIN** (**INNER**, **LEFT OUTER**, **RIGHT OUTER** e **FULL OUTER**) podem ser combinados com qualquer condição de **JOIN** (**NATURAL**, **USING** ou **ON**).

Join types
inner join
left outer join
right outer join
full outer join

Join conditions
natural
on <predicate>
using $(A_1, A_2, \ldots, A_n)$

Figura 2.2: Tipos e Condições de **JOIN**

Por fim, deixo uma nota sobre o **CROSS JOIN**. Estas 2 consultas são equivalentes:

```
01 |  SELECT *
02 |  FROM student, takes;
03 |
04 |  -- equivalente a
05 |
06 |  SELECT *
07 |  FROM student CROSS JOIN takes;
```

2.10 Vistas

Podemos querer criar uma coleção personalizade de relações "virtuais"[1] que melhor se adequam a uma certa intuição do utilizador sobre a estrutura da organização. No exemplo da universidade, podemos querer listar todos as secções de curso oferecidas pelo departamento de Física no 2º semestre do ano de 2016, com o edifício e número de sala de cada secção. A consulta correspondente seria:

```
01 |  SELECT course.course_id, sec_id, building, room_number
02 |  FROM course, section
03 |  WHERE course.course_id = section.course_id
04 |      AND course.dept_name = 'Fisica'
05 |      AND section.semester = 2
06 |      AND section.year = 2017;
```

[1] Diz-se virtual pois as vistas mapeiam dados das tabelas do **modelo físico** para um novo **modelo lógico**, havendo independência lógica entre estes 2 modelos.

É possível computar e guardar os valores destas consultas e torná-las disponíveis aos utilizadores. Porém, se algum dos valores em *instructor*, *course* ou *section* mudarem, os valores consultados guardados não correspondem à realidade.
Assim, o SQL permite uma "relação virtual"ser definida por uma consulta, e esta mesmo relação conceptual contém o resultado da consulta - a consulta é computada sempre que a relação virtual é usada. Já vimos algo semelhante com a clásula **WITH**, que nos permite nomear uma sub-consulta para uma consulta apenas. A **VIEW** permite estender o conceito de relação virtual para além de uma única consulta.

2.10.1 Definição de Vista

Definimos uma vista em SQL usando o comando **CREATE VIEW**. Para definir a vista devemos nomeá-la e definir qual a consulta que a computa.

```
01 |  CREATE VIEW v AS <expressao da consulta>;
```

Para a remover da base de dados, é igual a qualquer outro objeto de uma base de dados:

```
01 |  DROP VIEW v;
```

Considerando a consulta que foi apresentada no início do sub-capítulo, podemos definir uma vista sobre essa consulta.

```
01 |  CREATE VIEW fisica_1semestre_2017 AS
02 |      SELECT course.course_id, sec_id, building, room_number
03 |      FROM course, section
04 |      WHERE course.course_id = section.course_id
05 |          AND course.dept_name = 'Fisica'
06 |          AND section.semester = 2
07 |          AND section.year = 2017;
```

2.10.2 Vistas em Consultas SQL

Uma vez definida uma vista, podemos usar o nome que lhe foi atribuída para nos referirmos à relação que a vista gera.
1) Seja a seguinte consulta: "Liste todos os cursos de Física oferecidos no 2º semestre do ano de 2017 que estão no edifício Watson).". Temos:

```
01 |  SELECT course_id
02 |  FROM fisica_1semestre_2017
03 |  WHERE building = 'Watson';
```

Naturalmente, podemos criar vistas que usem vistas na sua consulta, desde que as vistas usadas na consulta estejam previamente definidas. Ou seja,

```
01 |  CREATE VIEW fisica_1semestre_2017_watson AS
02 |      SELECT course_id, room_number
03 |      FROM fisica_1semestre_2017
04 |      WHERE building = 'Watson';
```

2.10.3 Vistas Materializadas

Certos SGBD permitem relações serem armazenadas, mas para tal, se relações que são usadas em vistas mudarem, a vista é mantida atualizada - tais vistas são chamadas **vistas materializadas**.
Isto permite que os resultados duma vista sejam armazenados na base de dados, permitindo consultas que usem a vista para potencialmente **correrem muito mais rapidamente**, pois usa resultados pré-computados, ao invés de recomputá-los.
Assim, se um tuplo *instructor* é adicionada à relação *instructor* e uma vista usa essa mesma relação, é necessário manter a vista atualizada - o processo de manter a **vista materializada** atualizada é designado **manutenção da**

vista. Este processo pode ocorrer imediatamente quando uma das relações da qual a vista depende é atualizada, ou de um modo preguiçoso, apenas quando a vista é acedida.
Aplicações que usam uma vista frequentemente ou que necessitam resposta rápida a certas consultas que computam agregações sobre relações grandes beneficiariam de uma **vista materializada**.
No contexto do PostgreSQL (SGBD da cadeira), eis os comandos:

```
01 | -- Criar vistas materializadas
02 | CREATE MATERIALIZED VIEW view_exemplo AS SELECT ...
03 |
04 | -- Criar tabelas materializadas
05 | CREATE TABLE table_exemplo AS SELECT ...
06 |
07 | -- Atualizacao de vista materializada (MANUAL)
08 | REFRESH MATERIALIZED VIEW view_exemplo
09 |
10 | -- Atualizacao de vista materializada (AUTOMATICO)
11 | CREATE UNIQUE INDEX idx_view_exemplo
12 | ON view_exemplo(atributo);
13 |
14 | REFRESH MATERIALIZED VIEW CONCURRENTLY view_exemplo;
```

2.10.4 Atualização de Vistas

Nem todas as vistas são atualizáveis diretamente a partir dos respetivos comandos de atualização, pois a vista pode depender de várias relações simultaneamente.
Em geral, no SQL uma vista diz-se **atualizável**, i.e., permite o uso dos comandos **UPDATE**, **INSERT** e **DELETE** sse:

- A cláusula **FROM** só tem 1 relação;
- A cláusula **SELECT** contém apenas nomes de atributos da relação e **não tem** quaisquer expressões, agregações ou especificações **DISTINCT**;
- Qualquer atributo na cláusula **SELECT** pode ser posto a **null**; i.e, não tem uma restrição **not null** e não faz parte da chave primária;
- A consulta não tem as cláusulas **GROUP BY** ou **HAVING**.

Contudo, as vistas podem também ser definidas com a opção **WITH CHECK** que aquando da tentativa de atualização da vista, se as restrições especificados na opção não forem verificadas, a atualização é descartada.

2.11 Restrições de Integridade

Definição 7 (Restrição de Integridade)**.** Uma restrição de integridade assegura que mudanças feitas à base de dados por utilizadores **autorizados** não resultam em perca de consistência de dados.

2.11.1 Restrições em 1 Relação

Como já vimos, a criar tabelas podemos ter instruções de restrições de integridade. Eis algumas delas:

- **PRIMARY KEY** (já vimos anteriormente)
- **NOT NULL**
- **UNIQUE**
- **CHECK**(<predicado>)

2.11.2 Restrição NOT NULL

Para certos atributos, o valor null pode ser desapropriado. Por exemplo, na relação *student* não faz sentido ter o atributo *name* a null, pois representa um estudante desconhecido. Assim, adicionaríamos ao criar a relação *student*:

```
CREATE TABLE student
    (...,
    name VARCHAR(20) NOT NULL,
    ...;
```

Esta restrição proíbe a inserção de um valor null para o atributo especificado, e é um exemplo de uma **restrição de domínio**.
De notar também que o SQL, por defeito, não permite valores null em atributos que constituam a **chave primária** de uma relação.

2.11.3 Restrição UNIQUE

A especificação **UNIQUE** forma uma super-chave - nenhuns 2 tuplos na relação podem ter os atributos especificados no **UNIQUE** iguais. Um exemplo de uma espeficação **UNIQUE** é dado por:

```
UNIQUE (name, age, gender)
```

2.11.4 Cláusula CHECK

Um uso comum da cláusula **CHECK** é assegurar que valores de atributos satisfazem condições especificadas. Por exemplo, usando valores da vida real, convém que um orçamento para um departamento seja um número real positivo. Temos então:

```
-- Podemos escrever no fim
CREATE TABLE department
    (dept_name VARCHAR(20),
     building VARCHAR(15),
     budget NUMERIC(12, 2),
    PRIMARY KEY (dept_name),
    CHECK (budget > 0));
-- Ou a declarar o atributo
CREATE TABLE department
    (dept_name VARCHAR(20),
     building VARCHAR(15),
     budget NUMERIC(12, 2) CHECK (budget > 0),
    PRIMARY KEY (dept_name));
```

A cláusula **CHECK** permite atribuir restrições ao domínio de um atributo de uma forma potente.

2.11.5 Integridade de Referenciamento

Por vezes, queremos assegurar que um valor que aparece numa relação para um dado conjunto de atributos também aparece num conjunto de atributos duma outra relação. Tais condições dizem-se ser as **restrições de intregidade de referenciamento**, e as **FOREIGN KEY** são uma forma dessas restrições onde os atributos referenciados formam uma **chave primária** na relação referenciada. Eis um exemplo:

```
-- Podemos escrever no fim
CREATE TABLE course
    (course_id VARCHAR(8),
     title VARCHAR(50),
     dept_name VARCHAR(20),
     credits NUMERIC(2, 0) CHECK (credits > 0),
    PRIMARY KEY (course_id),
    FOREIGN KEY (dept_name) REFERENCES department);
-- Ou a declarar o atributo
CREATE TABLE course
    (course_id VARCHAR(8),
     title VARCHAR(50),
     dept_name VARCHAR(20) REFERENCES department,
     credits NUMERIC(2, 0) CHECK (credits > 0),
    PRIMARY KEY (course_id));
```

Esta declaração **FOREIGN KEY** exige que para cada tuplo em *course*, o nome do departamento especificado tem de existir na relação *department*.
Quando uma restrição de integridade de referenciamento é violada, o procedimento normal é rejeitar a ação que causou o violamento. Contudo, uma cláusula **FOREIGN KEY** pode especificar que se uma ação de remoção ou atualização na relação referida for violada, o sistema deve seguir alguns passos para mudar o tuplo na relação em que a restrição é definida para restaurar a mesma. Seja:

```
01 |  CREATE TABLE course
02 |      (...
03 |      FOREIGN KEY (dept_name) REFERENCES department
04 |                      ON DELETE CASCADE
05 |                      ON UPDATE CASCADE,
06 |      ...);
```

A cláusula **ON DELETE CASCADE** fará com que se removermos um tuplo na relação *department* que resulte na violação desta restrição de integridade na relação *course*, o sistema "por cascata"apaga os tuplos em *course* que referenciavam aquele nome de departamento. O comportamento do **ON UPDATE CASCADE** é análogo. Se quisermos outro comportamento em vez de ser apagar o tuplo em *course*, podemos definir o valor a null, metendo a cláusula **SET NULL** no lugar de **CASCADE**.

2.11.6 Restrições Nomeadas

É possível denominar uma restrição de integridade. Pode ser útil se quisermos remover uma restrição que fora definida previamente. Para tal, temos o exemplo:

```
01 |  -- Criar a restricao nomeada
02 |  CREATE TABLE instructor
03 |      (...
04 |      salary NUMERIC(8, 2), CONSTRAINT minsalary CHECK (salary > 1000),
05 |      ...);
06 |  -- Remover a restricao
07 |  ALTER TABLE instructor DROP CONSTRAINT minsalary;
```

2.12 Autorização

Podemos atribuir a um utilizador várias formas de autorização em partes da base de dados.
Para criarmos/removermos um utilizador usamos os seguintes comandos:

```
01 |  -- Criar um utilizador
02 |  CREATE USER 'Aragonez' WITH PASSWORD 'Joao';
03 |  -- Remover um utilizador
04 |  DROP USER [IF EXISTS] 'Aragonez';
```

Os vários tipos de autorização chamam-se **privilégios** e estes são:

- Autorização para ler dados;
- Autorização para inserir novos dados;
- Autorização para atualizar dados;
- Autorização para remover dados.

Quando um utilizador submete uma operação de consulta ou atualização, a implementação SQL verifica primeiramente se a operação é autorizada, com base nos privilégios do utilizador - se não for autorizada, é rejeitada.

2.12.1 Concessão e Revogação de Privilégios

O SQL *standard* inclui os **privilégios SELECT, INSERT, UPDATE** e **DELETE**. Inclui ainda o privilégio **ALL** que é um atalho para todos os privilégios anteriores.
A DDL do SQL inclui comandos para conceder e revogar privilégios. A instrução **GRANT** é usada para conferir autorização e tem a seguinte forma geral:

```
GRANT <lista de privilegios>
ON <nome da relacao ou vista>
TO <lista de utilizadores/papeis>
```

A autorização **SELECT** numa relação é necessária para ler tuplos na relação. Um exemplo de querermos dar ao utilizador "Aragonez"(utilizador de escolha daqui em diante) autorização para fazer consultas **SELECT** na relação *department* é dado por:

```
GRANT SELECT
ON department
TO Aragonez
```

A autorização **UPDATE** numa relação permite o utilizador atualizar qualquer tuplo na relação - podem ser dado todos os atributos da relação ou apenas alguns. No caso de selecionarmos apenas alguns atributos, devem ser seguidos da cláusula **UPDATE** um conjunto de parêntes ( ) com os atributos lá dentro especificados. Por exemplo:

```
GRANT UPDATE (budget)
ON department
TO Aragonez
```

A autorização **INSERT** numa relação permite o utilizador inserir tuplos na relação - podem ser dado todos os atributos da relação ou apenas alguns. No caso de selecionarmos apenas alguns atributos, devem ser seguidos da cláusula **INSERT** um conjunto de parêntes ( ) com os atributos lá dentro especificados. Para os valores que não pertençam a este conjunto, o sistema ou lhes atribui um valor **DEFAULT** se assim o for especificado, ou mete-os a **NULL**.

```
GRANT INSERT (budget)
ON department
TO Aragonez
```

A autorização **DELETE** numa relação permite o utilizador remover tuplos duma relação.

```
GRANT DELETE
ON department
TO Aragonez
```

O utilizador *public* refere-se ao utilizador atual e todos os futuros utilizadores do sistema - os seus privilégios são-lhes concecidos implicitamente.
O mecanismo de autorização do SQL concede privilégios numa relação inteira, ou em atributos específicos de uma relação. Contudo, não permite autorizações em tuplos específicos de uma relação.
Para revogar uma autorização, usamos a instrução *REVOKE*, que tem uma forma geral análoga à **GRANT**:

```
REVOKE <lista de privilegios>
ON <nome da relacao ou vista>
TO <lista de utilizadores/papeis>
```

Para revogarmos os privilégios concedidos previamente escrevemos:

```
REVOKE SELECT ON department FROM Aragonez;
REVOKE UPDATE (budget) ON department FROM Aragonez;
REVOKE INSERT (budget) ON department FROM Aragonez;
REVOKE DELETE ON department FROM Aragonez;
```

2.12.2 Papéis

Consideremos os papéis das várias pessoas no mundo real. Seja um papel por exemplo um instrutor, professor-assistente e estudante exemplos de papéis num contexto universitário.

Quando um novo instrutor for contratado, um identificador de utilizador terá de ser alocado a ele, e este deve ser identificado com o papel de instrutor, que engloba em si um conjunto de permissões no sistema.
A noção de **papéis** em SQL captura este conceito - um conjunto de papéis é criado na base de dados, e autorização são concedidas a papéis da mesma forma que são concedidas a utilizadores.
É verdade que poderíamos criar um único utilizador *instrutor* e todos acederem a partir desse mesmo, mas por questões de segurança, é preferível identificar exatamente que instrutor executou uma certa operação sobre a base de dados.
Os **papéis** em SQL são criados da seguinte maneira:

```
01 |  -- Criar o papel
02 |  CREATE ROLE instructor;
03 |
04 |  -- Conceder permissoes
05 |  GRANT SELECT
06 |  ON takes
07 |  TO instructor;
08 |
09 |  -- Papeis podem ser concedidos a utilizadores bem como a outros papeis
10 |  CREATE ROLE dean;
11 |  GRANT
12 |  instructor
13 |  TO dean;
14 |
15 |  GRANT UPDATE
16 |  ON takes
17 |  TO dean;
18 |
19 |  GRANT dean TO Aragonez;
```

Os privilégios de um utilizador/papel consistem em:

- Todos os privilégios concedidos diretamente ao utilizador/papel;
- Todos os privilégios concedidos aos papéis que foram concedidos ao utilizador/papel.

Isto pode criar correntes de papéis, o que deve ser gerido cuidadosamente.

2.12.3 Autorização em Vistas

Como vimos, as vistas também podem ser alvo de autorização.
Porém, é importante notar que um utilizador que cria uma vista não recebe necessariamente todos os privilégios da vista - só recebe os privilégios que não requiram autorização adicional para além do que o que já possui. Por exemplo, um utilizador que cria uma vista não pode ter a autorização **UPDATE** numa vista sem tem autorização **UPDATE** em todas as relações que a vista refere - se o utilizador criar uma vista sem permissões suficientes para tal, o sistema nega-lhe o pedido de criação de vista.

2.12.4 Autorização em Esquemas

O SQL *standard* especifica um mecanismo de autorização primitivo para o esquema de base de dados: apenas o **dono do esquema** pode modificá-lo, como criar e remover **relações**, adicionar ou tirar **atributos de relações**, bem como adicionar ou tirar **índices**.
Contudo, o SQL fornece um privilégio **REFERENCES** que permite a um utilizar declarar chaves secundárias ao criar relações. O privilégio é concedido a atributos específicos como o privilégio **UPDATE**. Passo a exemplificar:

```
01 |  GRANT REFERENCES (dept_name) ON department TO Aragonez;
```

2.13 Funções e Procedimentos

2.13.1 Introdução

Definição 8 (Base de Dados Centrada Em Dados)**.** O desenvolvimento de uma base de dados centrada em dados é uma abordagem de desenvolvimento onde instruções procedimentais (como um sub-programa em qualquer

linguagem de programação) pode ser armazenado dentro de uma base de dados também conhecidas como **stored program** / **stored routine**.

Um **Persistent Stored Module** (**PSM**) pode ser de 2 tipos:

- Uma **stored function** que **retorna um valor** mas **não altera** o estado da base de dados - combina *inputs* e resultados das consultas da base de dados para produzir o *output*.
- Um **stored procedure** que **pode mudar** o estado da base de dados e **não retorna nenhum valor** - consulta e atualiza a base de dados.

As vantagens de usarmos **PSM**:

- Torna as aplicações **mais rápidas**: os PSM são compilados e mantidos dentro da base de dados.
- **Reduz a troca de dados**: especialmente entre a aplicação e o servidor da base de dados.
- Introduz **1 nível de indireção**: pode ser chamado por aplicações escritas em linguagens distintas.

As desvantagens de usarmos **PSM**:

- **Debugging** e **profiling** pode ser extremamente difícil;
- O ambiente de desenvolvimento depende altamente do SGBD usado.

Funções e procedimentos permitem que a "lógica de negócio"seja guardada na base de dados e executada a partir de instruções de SQL. Por exemplo, universidades têm várias regras sobre quantas cadeiras um estudante por detirar ao mesmo tempo num dado semestre.

2.13.2 Declaração e Invocação de Funções e Procedimentos SQL

Para definirmos/removermos um procedimento ou uma função:

```
01 |  -- Criar
02 |  CREATE [OR REPLACE] PROCEDURE/FUNCTION ...
03 |  -- Remover
04 |  DROP PROCEDURE/FUNCTION ...
```

Para invocarmos um procedimento ou uma função (podem ser chamados a partir de outros procedimentos ou de aplicações a nível do cliente):

```
01 |  -- Chamar procedimento
02 |  CALL procedure_name [(param1, param2, ...)]
03 |  -- Chamar funcao
04 |  my_function_name([param1, param2, ...])
```

1) Seja a seguinte operação: "Cria um procedimento que insere o tipo de gomas preferidas do João Aragonez na relação *aragonez_senior_shopping_list* mediante o seu grau de fome ($< 50\%$ são as gomas ácidas, caso contrário são as doces)". Temos:

```
01 |  CREATE PROCEDURE fome_aragonez(IN grau_fome INTEGER) AS
02 |
03 |  DECLARE gomas VARCHAR(6);
04 |  BEGIN
05 |      IF grau_fome < 50
06 |      THEN
07 |          SET gomas := 'acidas';
08 |      ELSE
09 |          SET gomas := 'doces';
10 |      END IF;
11 |      INSERT INTO aragonez_senior_shopping_list
12 |          VALUES (gomas);
13 |  END
```

É de notar que o PostgreSQL não suporta **stored procedures** verdadeiros (um **PROCEDURE** em PostgreSQL é uma **FUNCTION** que retorna o *type* **VOID**); apenas **stored functions** são suportadas.
As **stored functions** na sua forma geral são:

```
CREATE FUNCTION my_func([parametros])
RETURNS type AS
$$
DECLARE [declaracoes]
BEGIN
    [instrucoes]
END
$$ LANGUAGE plpgsql;
```

De notar que o **PSM** no Postgres permite o *type* ser **VOID**.
2) Seja a seguinte operação: "Cria uma função que recebe 2 *numeric*'s como *input* e que retorna outro *numeric* correspondente à soma dos dois.". Temos:

```
CREATE FUNCTION add_nums(
    x NUMERIC,
    y NUMERIC)
    RETURNS NUMERIC AS
$$
BEGIN
    RETURN x + y;
END
$$ LANGUAGE plpgsql;
```

Podemos usar as **stored functions** nas cláusulas **SELECT** e **WHERE**. Exemplificando:

```
-- Sem clausula FROM
SELECT add_me(2, 3);

-- Como uma expressao no SELECT (aplicado a cada tuplo de instructor)
SELECT name, add_me(salary, 100)
FROM instructor

-- Como uma expressao no WHERE (aplicado a cada tuplo de instructor)
SELECT *
FROM instructor
WHERE add_me(salary, -1000) < 1000
```

Definição 9 (Função Determinística)**.** Uma função é determinística se produz sempre o mesmo resultado para os mesmos valores de *input*.

Em PostgreSQL, as função são **não-determinísticas** por defeito, pelo que se quisermos especificar que a função é determinística devemos usar o seguinte comando:

```
CREATE FUNCTION myfunc([params])
RETURNS type IMMUTABLE
```

As funções e procedimentos permitem a **declaração de variáveis**, como já vimos, sem complexidade alguma. A sua forma geral é dada por:

```
DECLARE var_name [, var_name2 ...] type [DEFAULT value]
```

Estas são apenas visíveis dentro do scope **BEGIN** ... **END**.
Um pequeno exemplo é dado por:

```
DECLARE
    first_name VARCHAR(50) DEFAULT 'Joao';
    last_name VARCHAR(50) DEFAULT 'Aragonez';
    contador INTEGER := 1; -- eh assim que se fazem atribuicoes em PostgreSQL
    pagamento NUMERIC(11, 2) := 20.5;
```

As variáveis podem também ser declaradas para captarem o resultado de uma consulta usando a cláusula **INTO**. Exemplificando a forma geral:

```
BEGIN
    ...
    SELECT col_name, ...
    INTO var_name
    FROM ...
    ...
END
```

É rápido para perceber que se quisermos vários atributos de uma relação guardados numa variável declarada, não existe nenhum *type* pré-definido que as albergue sem problemas. Para isso, podemos criar o nosso próprio tipo, representado em baixo um exemplo de um tipo relevante:

```
CREATE TYPE gomas_aragonez AS (
    tipo VARCHAR(6),
    preco_max NUMERIC(1, 2),
    perc_acucar NUMERIC(3, 2)
);
```

De notar que esta criação de tipos permite também ser fornecido como *type* de retorno de uma **stored function**.

2.13.3 Construtores de Linguagem para Funções e Procedimentos

Existem várias instruções dentro do scope **BEGIN** ... **END** da função/do procedimento.
A sintaxe para instruções **WHILE** e **REPEAT** é dada por:

```
-- Ciclo WHILE
WHILE expressao booleana DO
    sequencia de instrucoes;
END WHILE

-- Ciclo REPEAT
REPEAT
    sequencia de instrucoes;
UNTIL expressao boleana
END REPEAT

-- As palavras-chave break, continue e return nos ciclos em PostgreSQL sao dados por
EXIT -- sai do loop
CONTINUE -- continua o loop
RETURN -- sai da funcao (e do loop)
```

Existe ainda um ciclo **FOR** que permite iterar sobre os resultados de uma consulta:

```
DECLARE n INTEGER DEFAULT 0;
FOR r AS
    SELECT budget FROM department
    WHERE dept_name = 'Musica'
DO
    SET n = n - r.budget
END FOR
```

A sintaxe para instruções **IF-THEN-ELSE** é dada por:

```
IF expressao booleana
    THEN instrucao ou instrucao composta
ELSE IF expressao booleana
    THEN instrucao ou instrucao composta
ELSE instrucao ou instrucao composta
END IF
```

Se quiser ter o **PSM** a retornar **relações inteiras ou parciais** basta usar na cláusula **RETURNS** a palavra-chave **SETOF <relação>** se a quisermos na íntegra ou **TABLE(atr_1 type(atr_1), ..., atr_m type(atr_m))** se quisermos uma tabela parcialmente, assumindo que $m < n$, onde n é o número de nomes de atributos diferentes na relação.

Para ter o **PSM** a retornar **apenas um tuplo da relação** basta declarar uma variável do seguinte modo:

```
...
RETURNS table_name AS
$$
DECLARE
    var_name table_name%ROWTYPE
```

2.13.4 Blocos DO

Os blocos **DO** executam uma função anónima (sem qualquer nome atribuído). Eis um exemplo:

```
DO $$
DECLARE total NUMERIC DEFAULT 0;
BEGIN
    SELECT sum(salary)
    INTO total
    FROM instructor;
    RAISE INFO 'Total salarios pagos a instrutores --> %', total;
END
$$;
```

2.14 Triggers

Definição 10. Um trigger é uma instrução que o sistema executa automaticamente aquando de uma modificação na base de dados (operações como **INSERT, UPDATE** ou **DELETE**).

Para definirmos um trigger devemos:

- Especificar quando é que o trigger é suposto ser executado. Isto decompõe-se num *evento* que causa o trigger a ser verificado e uma *condição* deve ser satisfeita para a execução do trigger proceder.
- Especificar as *ações* a tomar quando o trigger executar.

Diz-se então que os triggers são especificados usando as regras ***event-condition-action***.

2.14.1 Necessidade de Triggers

Os triggers podem ser usados para:

- Implementar certas **restrições de integridade**;
- **Agir automaticamente** mediante a junção de certas condições bem definidas - estas ações podem ser atualizar/mudar/replicar relações, tirar medidas de estatística, etc.

2.14.2 Triggers em SQL

Como sabemos, um trigger é um procedimento que é automaticamente invocado em resposta a certas atualizações da base de dados.
A sua sintaxe em SQL é dada por:

```
-- Criar um trigger
CREATE TRIGGER <nome_trigger>
    { BEFORE | AFTER } { INSERT | UPDATE | DELETE }
ON <nome_relacao ou nome_vista>
WHEN <condicao>
FOR EACH { ROW | STATEMENT } EXECUTE PROCEDURE <nome_proc>

-- Remover um trigger
DROP TRIGGER <nome_trigger> ON <nome_relacao ou nome_vista> [IF EXISTS]
```

1) Seja a seguinte operação: "Cria um trigger que não permite que o salário de um instrutor seja abaixo de 800 nem acima de 3000.". Temos:

```
01 | -- Criar o procedimento a executar aquando de um trigger
02 | CREATE OR REPLACE FUNCTION verf_salario_proc()
03 | RETURNS TRIGGER AS
04 | $$
05 | BEGIN
06 |     IF NEW.balance < 800 THEN
07 |         NEW.balance := 800;
08 |     ELSEIF NEW.balance > 3000 THEN
09 |         NEW.balance := 3000;
10 |     ENDIF;
11 |     RETURN NEW;
12 | END;
13 | $$ LANGUAGE plpgsql;
14 |
15 | -- Definir o trigger
16 | CREATE TRIGGER verf_salario_trigger
17 |     BEFORE INSERT
18 |     ON instructor
19 |     FOR EACH ROW
20 |     EXECUTE PROCEDURE verf_salario_proc();
```

O comportamento dos triggers mediante serem **BEFORE** ou **AFTER** da operação que modifica a base de dados é resumido em:

	BEFORE	AFTER
ACTIVATION	O trigger é ativado 1 vez para cada tuplo ou instrução.	O trigger é ativado 1 vez para cada tuplo ou instrução sse: - Todos os triggers BEFORE na mesma relação e os relativos à mesma operação são executados com sucesso; - A operação foi executada com sucesso.
FAILURE	A operação na fila/relação não é executada. Nenhum trigger AFTER é ativado.	A operação na fila/relação não é executada.

2.14.3 Problemas dos Triggers

Os triggers têm *um efeito complexo*, por vezes imprevisível, pois vários triggers podem serem ativados numa única operação e a ação de um trigger pode ativar outro (**triggers recursivos**).
Podem, como tal, gerar *ciclos de eventos*, com muita dificuldade em depurar e corrigir, bem como instanciar *execuções indesejadas* pois mudanças a uma tabela podem gerar triggers por cadeia que não eram intencionados serem executados.
Se *ocorrerem erros* e o trigger falhar, a operação inteira falha, e é complexa a sua recuperação de falta.

2.14.4 Quando Não Usar Triggers

- *Tabelas de Resumo*: usar **VIEW**s em vez de **TRIGGER**s, se possível;
- *Restrições de Integridade Complexas*: usar **CHECK**s sempre que possível;
- *Replicação de Relações*: usar os mecanismos embutidos do SGBD.

Capítulo 3

Desenvolvimento de Aplicações com Bases de Dados

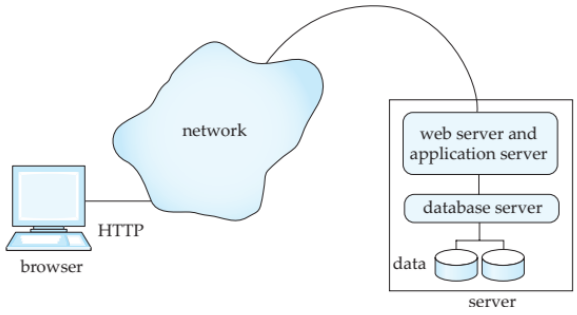
A maneira mais comum dos utilizadores interagirem com bases de dados é através de *programas aplicacionais* que fornecem uma interface ao utilizador no *frontend* e interfaces com a base de dados no *backend*.

3.1 Programas Aplicacionais e Interfaces de Utilizador

Um *programa aplicacional* típico contém uma componente *frontend*, que lida com a interface do utilizador, uma componente *backend*, que comunica com a base de dados, e uma *middle layer*, que contém lógica de negócio, i.e., código que executa pedidos específicos para informação ou atualizações - os comandos SQL podem ser invocados através de qualquer aplicação informática, escrita em qualquer linguagem (*Java, C, PHP, Python, ...*).
Para evitar acesso direto do utilizador à base de dados, que compromete a segurança da mesma, e para haver um único ponto de mudança, i.e., se um utilizador fizer uma mudança local, esta está automaticamente disponível aos demais utilizadores da aplicação e estes não precisarem de descarregar/atualizar a aplicação localmente, usam-se 2 abordagens:

- *Web browsers* oferecem uma ***frontend* universal** - usam HTML (**linguagem *markup***) como sintaxe e apresentam o conteúdo das páginas de forma "gráfica"ao utilizador.
- Os programas são instalados localmente, mas comunicam com as aplicações *backend* através de uma **API** e não têm acesso direto à aplicação - é mais usada em aplicações móveis e o seu resultado é normalmente **XML**.

O modelo de arquitetura de aplicações *web* lecionado na cadeira é o seguinte:



(a) Arquitetura Aplicação Web

(b) Distribuição Física

Figura 3.1: Aplicações Web em BD

3.2 Acesso a SQL numa Linguagem de Programação

Existem 2 abordagens para aceder a SQL numa linguagem de programação:

- **SQL dinâmico**: permite o programa construir consultas SQL como uma cadeia de caracteres em *runtime*, submeter a consulta e colocar o resultado em variáveis do programa, um tuplo de cada vez.
- **SQL embutido**: as instruções SQL são identificadas em tempo de compilação usando um pré-processamento que traduz os pedidos feito em SQL embutido em chamadas de funções.

3.2.1 SQL Embutido

Uma linguagem na qual consultas SQL são embutidas é referida como a *linguagem host*, e as estruturas SQL permitidas na *linguagem host* constituem o **SQL embutido**.
Programas escritos na *linguagem host* podem usar SQL embutido para aceder e atualizar dados guardados na base de dados - existe um pré-processamento feito antes da compilação que transforma pedidos SQL embutidos com declarações na *linguagem host* e chamadas a procedimentos que permitem o acesso à base de dados durante o *runtime*.
Para identificar pedidos SQL embutidos ao pré-processador, usamos a instrução **EXEC SQL**, com a forma geral:

```
EXEC SQL <instrucao SQL embutida>;
```

Inicialmente, o programa deve-se conectar à base de dados:

```
-- Para o exemplo iremos usar C como linguagem host
-- Declaracao de variabeis
char* gomas = "Gomas! Yummy!";
EXEC SQL BEGIN DECLARE SECTION
char* username = "Aragonez";
char* password = "Joao";
EXEC SQL END DECLARE SECTION
-- Conexao ah base de dados
EXEC SQL CONNECT :username IDENTIFIED BY :password
-- Codigo C
printf("%s\n", gomas);
```

Como vimos, as variáveis da *linguagem host* podem ser usadas em instruções de SQL embutido, mas têm de ser precedidos por dois pontos (:), para as distinguir das variáveis do SQL embutido (**SQLCODE** que é um *long* que retorna negativo se existir um erro e **SQLSTATE** que é uma cadeia de caracteres que indica o tipo de erro).
Para iterar sobre os resultados de uma consulta SQL embutido, devemos declarar uma variável **CURSOR**, que pode ser aberto, e buscar comandos emitidos num loop da *linguagem host* para buscar linhas consecutivas do resultado da consulta - atributos da linha podem ser metidas em variáveis da *linguagem host*.
Exemplificamos agora com um programa em C que usa SQL embutido:

```
-- Declaracao de variabeis
char SQLSTATE[6]; -- variavel do SQL embutido
EXEC SQL BEGIN DECLARE SECTION
char c_iname[30];
short c_minsalary;
float c_iage;
EXEC SQL END DECLARE SECTION
c_minsalary = random();
-- Declarar o cursor para a consulta SQL embutida
EXEC SQL DECLARE iinfo CURSOR FOR
    SELECT name, age
    FROM instructor
    WHERE salary > :c_minsalary
    ORDER BY name;
-- Ciclo na linguagem host para dar fetch
do {
    EXEC SQL FETCH iinfo INTO :c_iname, :c_age; -- dar fetch p variaveis locais
    printf("%s is %d years old!\n", c_iname, c_age); -- codigo C puro
} while (SQLSTATE != '02000');
EXEC SQL CLOSE iinfo; -- fechar o cursor
```

Podemos ainda usar a instrução SQL **PREPARE** para executar instruções de SQL embutido pré-definidas, como por exemplo:

```
01 | -- Variavel da linguagem host com instrucao SQL embutida definida
02 | char c_sqlstring[] = {"DELETE FROM instructor WHERE age > 65"};
03 | -- Correr a instrucao da variavel
04 | EXEC SQL PREPARE foobar FROM :c_sqlstring
05 | EXEC SQL EXECUTE foobar
```

3.3 Programação de Aplicação Web

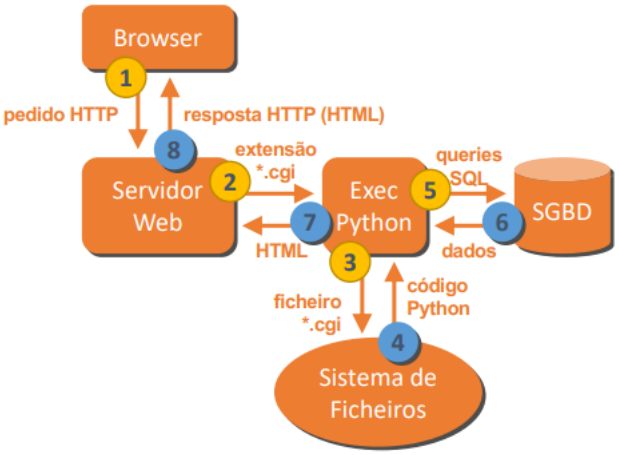
A **Common Gateway Interface (CGI)** define como é que um servidor web comunica com programas aplicacionais. Por sua vez, os programas aplicacionais comunicam com um servidor de base de dados atráveis de **ODBC/JDBC**.
Como o protocolo **HTTP** é *connectionless* não existe ligação contínua entre o cliente e o servidor web - quando o servidor recebe um pedido, uma conexão é temporariamente criada que permite a troca de mensagens entre o cliente e o servidor.
Em contraste, quando um utilizador se liga a uma base de dados usando textbfODBC/JDBC, uma sessão é criada e até esta ser terminada toda a informação de sessão é mantida no servidor.
Como as aplicações web (na sua maioria) necessitam de informação de sessão para permitir interagir eficientemente com o utilizador, utilizam-se **cookies** (já se viu isto em RC, não é relevante explicar como funcionam).

3.3.1 Introdução a Scripting, Framework e Driver

Usamos ***scripting languages*** e ***application frameworks*** para processar pedidos no servidor aplicacional.
Em **server-side scripting**, antes de entregar uma página *Web* ao cliente, o servidor executa os *scripts* embutidos nos conteúdos HTML da página. Cada pedaço do *script*, quando executado, pode gerar texto que é adicionado à página (ou até mesmo apagar). O código-fonte é apagado da página, pelo que o cliente poderá nem saber que havia código nele antes. O *script* executado pode conter código SQL que é executado sobre uma base de dados, bem como usar diretamente valores de input de formulários HTML preenchidos pelo cliente nas consultas SQL.
Em **client-side scripting**, os browsers conseguem buscar certos scripts do lado do cliente, ou até mesmo programas ou documentos, e executá-los de um "modo seguro"no lado do cliente, como é exemplo de certos scripts *Javascript*, *Adobe Flash* usado para animações/jogos, etc.
O Python oferece 2 frameworks para *server-side scripting*: **Flask** e *Django*. Oferece ainda um *driver* como modelo de acesso a SGBD: **Psycopg**.

3.3.2 Flask & Psycopg

Deixo aqui um fluxo da execução de um servidor Web que opera em conjunto com uma SGBD:



A figura acima pode ser legendada da seguinte forma:

1. O cliente realiza um pedido através do protocolo HTTP ao servidor Web;
2. O servidor Web tenciona executar um programa externo (*script CGI*) escrito numa *scripting language* (Python neste caso);
3. O servidor Web acede ao sistema de ficheiro local para encontrar o *script* localmente;
4. O sistema de ficheiros fornece o código Python para ser executado.
5. Quaisquer consultas SQL que estejam no código são agora realizadas, havendo comunicação com o SGBD;
6. O SGBD retorna os resultados das consultas;
7. Os resultados são transformados em HTML e o *server-side script* é apagado da página fonte;
8. A página é devolvida ao cliente.

Escrevamos agora o nosso *script* usando o framework **Flask** e o driver **Psycopg**:

```
01 | #!/user/bin/python3 ## IMPRESCINDIVEL
02 | from wsgiref.handlers import CGIHandler
03 | from flask import Flask
04 | import psycopg2
05 | import psycopg2.extras
06 |
07 | ## Configuracao do SGBD (Psycopg)
08 | DB_HOST = "db.tecnico.ulisboa.pt"
09 | DB_USER = "aragonez"
10 | DB_DATABASE = "aragonez_db"
11 | DB_PASSWORD = "joao"
12 | DB_CONNECTION_STRING = "host=%s dbname=%s user=%s password=%s" % (
13 |     DB_HOST,
14 |     DB_DATABASE,
15 |     DB_USER,
16 |     DB_PASSWORD,
17 | )
18 |
19 | ## Configuracao do Framework (Flask)
20 | app = Flask(__name__)
21 | ## Pagina principal mostra o index
22 | @app.route("/") ## Decorator - adiciona funcionalidade ao codigo existente
23 | def index():
24 |     try:
25 |         return render_template("index.html") ## render_template built-in do Flask
26 |     except Exception as e:
27 |         return str(e)
28 |
29 | ## Criar uma categoria
30 | @app.route("/criar_categoria", methods=["POST"]) ## POST pq o cliente enviou dados ao
      servidor atraves do form
31 | def executar_criar_categoria():
32 |     dbConn = None
33 |     cursor = None
34 |     try:
35 |         ## ligar ah bd
36 |         dbConn = psycopg2.connect(DB_CONNECTION_STRING)
37 |         cursor = dbConn.cursor(cursor_factory=psycopg2.extras.DictCursor)
38 |         ## processamento do input do cliente
39 |         name = request.form['category_to_add']
40 |         if (request.form['category_type'] == 'Simple Category'):
41 |             query = "INSERT INTO categoria_simples VALUES (%s);"
42 |         else:
43 |             query = "INSERT INTO super_categoria VALUES (%s);"
44 |         cursor.execute(query, (name, ))
45 |         query = "INSERT INTO categoria VALUES (%s);"
46 |         cursor.execute(query, (name, ))
47 |         return query ## Retorna o resultado da query como string para o HTML
48 |     except Exception as e:
49 |         return str(e)
50 |     finally:
51 |         dbConn.commit()
```

```
52 |           cursor.close()
53 |           dbConn.close()
54 |
55 |   ## Listar os produtos
56 |   @app.route('/listar_produtos')
57 |   def listar_produtos():
58 |       dbConn = None
59 |       cursor = None
60 |       try:
61 |           dbConn = psycopg2.connect(DB_CONNECTION_STRING)
62 |           cursor = dbConn.cursor(cursor_factory=psycopg2.extras.DictCursor)
63 |           query = "SELECT * FROM product;"
64 |           cursor.execute(query)
65 |           return render_template("listar_produtos.html", cursor=cursor)
66 |       except Exception as e:
67 |           return str(e)
68 |       finally:
69 |           dbConn.commit()
70 |           cursor.close()
71 |           dbConn.close()
72 |
73 |   ## IMPERATIVO ESTAR PRESENTE
74 |   CGIHandler().run(app)
```

Na função em que listamos os produtos, o código chama a função *render_template* embutida no próprio **Flask**. Este procura os *templates* pelo nome do ficheiro dado na pasta *templates*.
O **Flask** por sua vez usa o **Jinja2** como seu motor de *templates*. Eis alguns dos **delimitadores** importantes do mesmo:

- {%...%} declarações, ex.: if, for, while;
- {{...}} para expressões a serem substituídas na página gerada;
- {#...#} para comentários que não devem ser incluídos na página gerada;
- # para uma linha que contém uma instrução

Exemplo simples da utilização destes delimitadores (*listar_produtos.html*):

```
01 |  <!doctype html>
02 |  <title> Listar Produtos - Flask</title> <body style="padding:20px">
03 |  {% if cursor %}
04 |      <table border="2px">
05 |      <thead> <tr>  <th>EAN</th> <th>Descricao</th> <th>Alterar</th> </thead>
06 |      <tbody>
07 |          {% for record in cursor %}
08 |              <tr> <td>{{record[0]}}</td> <td>{{record[1]}}</td> <td><a href="description?
       product_ean={{record[0]}}">Alterar Descricao</a> </td> </tr>
09 |          {% endfor %}
10 |      </tbody> </table>
11 |  {% else %}
12 |      <p> Erro: Falha a obter dados da base de dados! </p>
13 |  {% endif %}
```

Se notarmos, a terceira entrada da tabela a partir da esquerda (começando em 1) permite alterar a descrição de um produto. Para isso usa um **href** que leva para uma nova página *description* e leva como argumento o EAN do produto em questão. Agora basta extender o script *CGI* em **Flask** para permitir realmente mudar a descrição:

```
01 |  ...
02 |
03 |  ## Configurar o routing para /description
04 |  @app.route("/description")
05 |  def change_description():
06 |      try:
07 |          ## params = request.args = records[0] passado no listar_produtos.html
08 |          return render_template("description.html", params=request.args)
09 |      except Exception as e:
10 |          return str(e)
```

Definimos agora o *template description.html*:

```
<html>
    <body>
        <h3>Alterar descricao do produto "{{ params.get("product_ean") }}"</h3>
        <!-- Falta-nos agora usar isto no Flask-->
        <form action="update_desc" method="post">
            <p><input type="hidden" name="ean" value="{{ params.get("product_ean")}}"/><
    /p>
            <p>Nova descricao: <input type="text" name="description"/></p>
            <p><input type="submit" value="Submeter"/></p>
        </form>
    </body>
</html>
```

E completamos por fim o *CGI*:

```
...

## Configurar o routing para /description
@app.route("/update_desc", methods=["POST"])
def update_description():
    dbConn = None
    cursor = None
    try:
        dbConn = psycopg2.connect(DB_CONNECTION_STRING)
        cursor = dbConn.cursor(cursor_factory=psycopg2.extras.DictCursor)
        query = f"UPDATE product SET description={request.form["description"]} WHERE ean
    ='{request.form["ean"]}';" ## SUJEITO A INJECAO SQL
        cursor.execute(query)
        return query
    except Exception as e:
        return str(e)
    finally:
        dbConn.commit()
        cursor.close()
        dbConn.close()
```

3.3.3 Injeção SQL

O problema com o código acima é que imaginando que no *input* do cliente na *description* do formulário HTML, este mesmo escreve:

```
'new_descr' WHERE ean='111'; UPDATE users SET admin = 'true' WHERE username = 'Aragonez';
```

Isto irá correr diretamente ao executarmos *cursor.execute(query)*, o que representa grandes riscos de segurança. Em **Psycopg**, a maneira correta de executar consultas SQL é da seguinte forma:

```
query = "UPDATE product SET description=%s WHERE ean=%s"
data = (request.form["description"], request.form["ean"])
cursor.execute(query, data)
```

Outras recomendações básicas de segurança para evitar injeções SQL são:

- *Forms* com *method = POST* (e não *GET*);
- Instruções **PREPARED**;
- SSL;
- Certificados.

Capítulo 4

Teoria da Normalização

Na conceção do modelo E-A, há certas decisões que se tomam que têm um grande impacto tanto ao nível lógico como ao nível físico e que se só são detetadas no modelo Relacionalm, obrigando por vezes a um processo iterativo de sucessivas reformulações do modelo E-A. Essas qualidades prendem-se com questões de **redundância e independência de atributos** das relações.

4.1 Redundância e Independência de Atributos

Em geral, a **redundância** consiste na existência de atributos em relações que se podem derivar diretamente de outros da mesma relação. Muitas vezes, esta redundância encontra-se ligada à inexistência de **dependência entre atributos**. Por isso, para além de causar desperdício de memória física, pode ter alguns efeitos indesejáveis, que se explicam de seguida.

Anomalias de Inserção

Por anomalias de inserção entende-se a impossibilidade de inserir um *item* numa base de dados sem inserir igualmente outro item **potencialmente independente/não relacionado** com este.

Redundancy

EMP_DEPT

Ename	Ssn	Bdate	Address	Dnumber	Dname	Dmgr_ssn
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Administration	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4	Administration	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975 FireOak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Research	333445555
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Headquarters	888665555

Considere-se uma base de dados que só contém a relação $EMP\_DEPT(Ename, Ssn, Bdate, Dnumber, Dname, Dmgr\_ssn)$ que indica que o funcionário com nome $Ename$, identificador Ssn, data de nascimento $Bdate$, morada $Address$ trabalha no departamento com número $Dnumber$, nome $Dname$ gerido pelo funcionário com identificador $Dmgr\_ssn$.
A inserção de um novo funcionário obriga a inserção dos dados do departamento no qual trabalha, levando potencialmente a entradas a **NULL** (indesejáveis). Mais ainda, a criação de um departamento ainda sem funcionários faria com que a informação relativa ao funcionário trabalhador ficasse por preencher. Desta forma, o atributo Ssn (chave primária da relação!) estaria a **NULL**.

Anomalias de remoção

Por anomalias de remoção entende-se a impossibilidade de remover um *item* numa base de dados sem remover igualmente outro item **potencialmente independente/não relacionado** com este.

Considerando ainda a base de dados anterior, a remoção do de um trabalhador único num dado departamento apagaria por completo qualquer informação relativa ao mesmo.

Anomalias de atualização

Por anomalias de atualização entende-se a impossibilidade de atualizar um *item* numa base de dados sem atualizar igualmente outro item **potencialmente independente/não relacionado** com este.
Por exemplo, a mudança do identificador do gestor de um dado departamento ($Dmgr\_ssn$) ter-se ia que fazer tantas vezes quantos funcionários trabalhassem nesse departamento.

Anomalias de interrogação

Dado que se usa mais memória nas tabelas manipuladas, verifica-se um maior tempo nas operações I/O e maior consumo de largura de banda, tornando obrigatoriamente as *queries* mais ineficientes.

4.2 Formas Normais

Por forma a caracterizar o grau de de redundância, independência dos factos e facilidade de interrogação, existem **Formas Normais** - classes de relações que obedecem a determinadas condições (kNF designa a k-ésima Forma Normal e $BCNF$ a Forma Normal de Boyce-Codd (*Boyce-Codd Normal Form*)). Verificam-se as seguintes inclusões:

$$1NF \subseteq 2NF \subseteq 3NF \subseteq BCNF \subseteq 4NF... \tag{4.1}$$

A maioria destas formas utiliza **Dependências Funcionais** entre atributos de uma relação como métrica de redundância.
Só serão abordadas as formas normais até à $BCNF$, pois esta é a última cujos critérios assentam sobre dependências funcionais.

4.3 Dependências Funcionais

Definição 11 (Dependência Funcional - FD)**.** Uma Dependência Funcional $X \to Y$ entre dois subconjuntos de atributos X e Y dos atributos R de uma relação r estabelece que (dados dois tuplos t_1 e t_2 de r):

$$\forall t_1, t_2 \in r,\ t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$$

Mais intuitivamente, os atributos de Y **são função** de X. Diz-se que X determina Y ou que X **determina funcionalmente** Y, sendo que X é *determinante* e Y *dependente*.

De notar que as dependências funcionais são propriedades um esquema de relação e não de um conjunto particular de registos. Desta forma, **é impossível** inferir dependências funcionais a partir de um conjunto de regitos, sendo unicamente possível provar que uma dada FD não existe através de um contraexemplo desse mesmo conjunto de registos (e.g., a existência de dois conjuntos de atributos Y associados a um mesmo conjunto de atributos X).

4.3.1 Propriedades

As seguintes propriedades designam-se de **Axiomas de Armstrong**:

$$Y \subseteq X \Rightarrow X \to Y \qquad \text{(reflexividade)}$$
$$X \to Y \Rightarrow XZ \to YZ,\ \forall Z \qquad \text{(aumentação)}$$
$$X \to Y \wedge Y \to Z \Rightarrow X \to Z \qquad \text{(transitividade)}$$

E as seguintes propriedades podem-se derivar dos axiomas:

$$X \to X \qquad \text{(reflexividade)}$$
$$X \to YZ \Rightarrow X \to Y \wedge X \to Z \qquad \text{(decomposição)}$$
$$X \to Y \wedge X \to Z \Rightarrow X \to YZ \qquad \text{(união)}$$
$$X \to Y \wedge A \to B \Rightarrow XA \to YB \qquad \text{(composição)}$$
$$X \to Y \wedge YZ \to W \Rightarrow X \to W \qquad \text{(pseudo-transitividade)}$$

4.3.2 *Attribute Closure*

Definição 12 (Fecho de um conjunto de atributos)**.** Seja X um subconjunto de atributos de um esquema relacional com um conjunto de dependências funcionais F. O fecho de X, X^+, consiste em todos os atributos que que **dependem funcionalmente** de X, isto é:

$$X^+ = \{Z \in R : F \models X \to Z\}$$

i.e., os conjuntos dos atributos Z de F tais que a dependência $X \to Z$ pode ser inferida a partir dos axiomas de Armstrong e propriedades derivadas.

Algoritmo 1 Algoritmo para calcular o *attribute closure*

```
X+ ← X
repeat
    X_old ← X+
    for each Y → Z ∈ F do
        if X+ ⊇ Y then
            X+ ← X+ ∪ {Z}
        end if
    end for
until X+ = X_old
```

Podemos então revisitar alguns conceitos à luz do *attribute closure*.

Definição 13 (Super-chave, chave candidate, chave primária)**.** Dada uma relação r com esquema R, e um subconjunto $K \subseteq R$, tem-se que:

1. K é uma **super-chave** de $r(R)$ se $K \to R$, ou, *analogamente*, $K^+ = R$;
2. K é uma **chave candidata** de $r(R)$ se e só se $K \to R \ \wedge \ \nexists(\alpha \subset K) : \alpha \to R$ (i.e., uma super-chave minimal: nenhum subconjunto de K determina funcionalmente R)

Sendo que uma **chave primária** é uma qualquer chave escolhida do conjunto de chaves candididates de R.

O seguinte conceito também revelar-se-á útil.

Definição 14 (Dependência total)**.** Sejam X e Y dois quaisquer conjuntos de atributos tais que $X \to Y$, diz se que Y é **totalmente dependente** em X se nenhum subconjunto próprio de X determina funcionalmente Y, i.e., $\nexists(\alpha \subset X) : \alpha \to Y$.

Nota: uma chave candidata é uma super-chave na qual R depende totalmente.

4.4 Descrição das Formas Normais

4.4.1 1ª Forma Normal

Definição

Uma relação encontra-se na 1ª forma normal se e só se:

- o domínio de todos os atributos contém apenas valores *atómicos* (i.e., indivisíveis);
- o valor num dado tuplo de uma relação tem de conter **um único valor** do domínio desse atributo.

Normalização-$1NF$

Uma relação que não se encontra na $1NF$ por apresentar domínios com valores não-atómicos pode ser normalizada ao transferir os campos do tuplo para uma nova relação, propagando a chave primária. Considere-se o seguinte exemplo com o atributo $Proj$ que é na verdade uma relação $Proj(Pnumber, Hours)$.

EMP_PROJ

		Projs	
Ssn	Ename	Pnumber	Hours

EMP_PROJ1

Ssn	Ename

EMP_PROJ2

Ssn	Pnumber	Hours

Se relação que não se encontrar na $1NF$ por apresentar atributos com listas de valores, a normalização ocorre fazendo um registo para cada valor do atributo composto, passando a chave primária do registo a incorporar o novo atributo:

(b)
DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)
DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Nota: uma relação que não se encontra na 1º forma normal não pode sequer ser considerada uma relação.

4.4.2 2ª Forma Normal

Definição

Uma relação encontra-se na 2ª forma normal se e só se se encontrar na 1º forma normal e **todo o atributo não chave** (i.e., que não participa em nenhuma chave candidata) é **completamente dependente** em **atributos chave** (i.e., depende sempre totalmente de *qualquer chave candidata*).

Normalização-$2NF$

A conversão de uma relação na 1ª forma normal para a 2ª forma normal faz-se do seguinte modo:

– Para cada atributo não chave R_n que **depende parcialmente** de um subconjunto de atributos chave de uma chave candidata K, $S \subseteq K$, retirá-lo da relação inicial R e criar uma nova relação com chave primária S e com esquema (S, R_n). Mais formalmente, ocorre a seguinte decomposição da relação inicial R:

$$R \leftarrow \{R \setminus R_n,\ (S, R_n)\}$$

afetando-se a cada relação resultante as FD's com cujos atributos constam nas relações resultantes.

Por exemplo, para a relação $R(\underline{A, B}, C, D)$ com chave primária $\{A, B\}$ e com dependências $AB \rightarrow C$ e $A \rightarrow D$, tem-se que D, não chave, depende parcialmente de uma chave (neste caso, da chave primária), dado que $A \rightarrow D$ e $A \subset \{A, D\}$. Assim, R decompõe-se em duas relações: $R_1(\underline{A, B}, C)$, com dependência $AB \rightarrow C$, e $R_2(\underline{A}, D)$, om dependência $A \rightarrow D$.

Nota: Uma relação na 1NF cujas chaves possuem um só atributo encontram-se trivialmente na 2NF.

4.4.3 3ª Forma Normal

Definição

Uma relação encontra-se na 3ª forma normal se e só se se encontrar na 2º forma normal e se e só se **não há dependências entre atributos não-chave**.

Normalização-$3NF$

A conversão de uma relação na 2ª forma normal para a 3ª forma normal faz-se do seguinte modo:

- Para cada atributo não chave R_n que depende funcionalmente num conjunto de atributos não-chave N, retirá-lo da relação inicial R e criar uma nova relação com chave primária N e com esquema (N, R_n). Mais formalmente, ocorre a seguinte decomposição da relação inicial R:

$$R \leftarrow \{R \setminus R_n,\ (N, R_n)\}$$

  afetando-se a cada relação resultante as FD's com cujos atributos constam nas relações resultantes.

Por exemplo, a relação $R(\underline{A}, B, C)$ com chave primária A e com dependências $B \rightarrow C$ tem-se que C, não chave, depende de B, um atributo não chave. Assim, R decompõe-se em duas relações $R_1(\underline{B}, C)$, com a FD $B \rightarrow C$, e $R_2(\underline{A}, B)$, com a FD $A \rightarrow B$.

Nota: Uma relação na 2NF que só possua um atributo naõ-chave encontra-se trivialmente na 3NF.

4.4.4 Forma Normal de *Boyce-Codd*

Definição

Uma relação encontra-se na Forma Normal de *Boyce-Codd* se e só se todo e qualquer atributo é totalmente dependente numa chave candidata:

$$\forall X \rightarrow Y, X \text{ é chave candidata}$$

excluindo obviamente as denominadas *dependências triviais*, i.e., da forma $X \rightarrow Y$, para algum $Y \subseteq X$.
Quando uma relaçaõ está na $BCNF$, então **não apresenta qualquer redundância** detedada por dependências funcionais.
Antes de expormos a passagem para a $BCNF$, vamos introduzir alguns conceitos relativos à decomposição de relações.

4.5 Decomposição de Relações

Para além das formas normais, é desejável que as relações das bases de dados apresentem propriedades relativas a interrelações entre estas, propriedades estas que devem ser preservadas na decomposição de relações que permite a passagem para formas normais de grau superior.
Dado um esquema de relação $R = (A_1, A_2, ..., A_n)$, pretende-se obter uma decomposição $D = \{R_1, ..., R_m\}$ que tenha as seguintes propriedades:

Propriedade da Preservação de Atributos

Pretende-se que cada atributo A_j figure em pelo menos uma relação R_i, i.e.:

$$\bigcup_{i=1}^{m} R_i = R$$

Propriedade da Preservação de Dependências

Dado que cada dependência representa um constragimento sobre a base de dados, convém preservá-las numa decomposição, i.e., para toda a dependência funcional, tem de existir **pelo menos uma relação** que contenha todos os atributos dessa dependência (assim, não há perda de dependências ao afetá-las a relações recém criadas - ver exemplos acima).

Propriedade *Lossless*

Informalmente, diz-se que uma decomposição é *lossless* (*não-aditiva*) se a aplicação de um *natural join* às relações relativas à decomposição resulta **exatamente** na relação inicial, não criando nenhum registo a mais.
Mais formalmente, uma decomposição $D = \{R_1, ..., R_m\}$ de R diz-se *lossless* em relação a um conjunto de FD's F de R se, para qualquer instância de relação r com esquema R, se tem $\pi_{R_1}(r) \bowtie ... \bowtie \pi_{R_m}(r) = r$, onde π designa o operador de projeção da álgebra relacional.

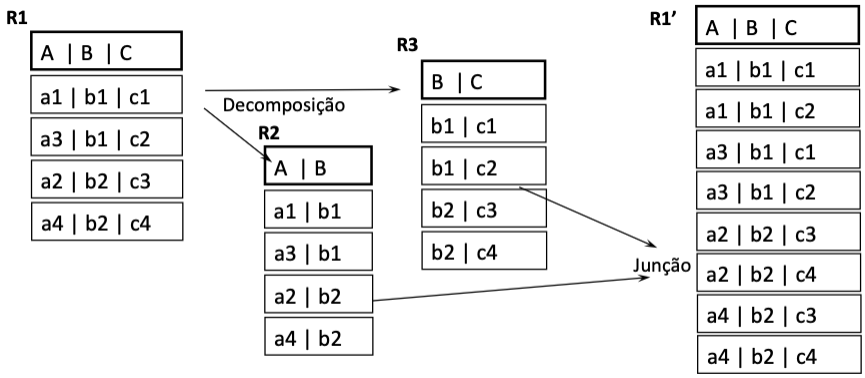


Figura 4.1: Decomposição *lossy*

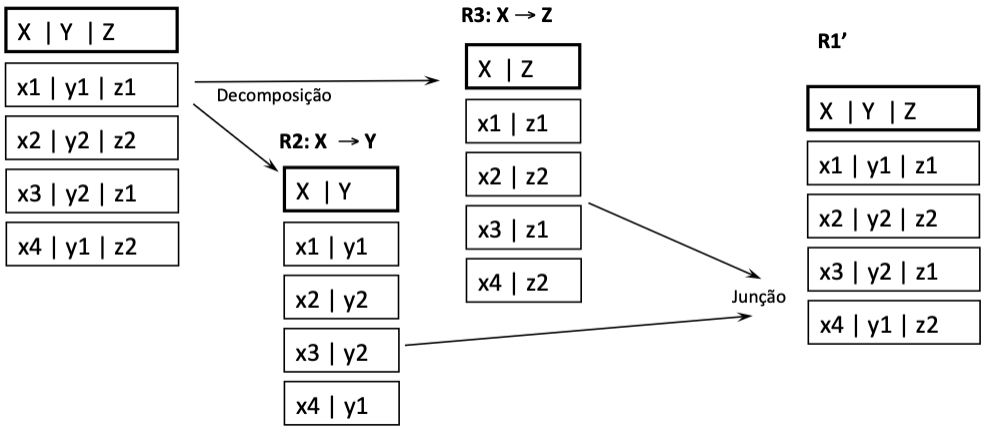


Figura 4.2: Decomposição *lossless*

Lema 1 (Sucessão de decomposições *lossless*)**.** *Se decomposições $D = \{R_1, R_2, ..., R_m\}$ e $D_i = \{Q_1, Q_2, ..., Q_m\}$ são lossless, então a decomposição $D_2 = \{R_1, R_2, ..., R_{i-1}, Q_1, Q_2, ..., Q_m, R_{i+1}, ..., R_m\}$ é também lossless.*

Teorema 1 (Teorema de *Heath*)**.** *Uma decomposição $D = \{R_1, R_2\}$ é lossless se pelo menos uma das duas dependências existir:*

1. $(R_1 \cap R_2) \rightarrow (R_1 \setminus R_2)$
2. $(R_1 \cap R_2) \rightarrow (R_2 \setminus R_1)$

Nota: Mesmo que se consiga uma decomposição lossless, nem sempre se consegue que as decomposições preservem dependências. Contudo, não se considera que tal seja de grande gravidade, pois pode se verificar a validade das dependências recorrendo a operações de join em troca de um maior esforço computacional. Assim, o critério a ter na escolha de uma decomposição é: dada uma decomposição obrigatoriamente lossless, escolhar aquela que perde o menor número de dependências.

Nota: Pode-se verificar que as decomposições feitas na normalização-1nf, normalização-2nf e normalização-3nf são lossless - basta atentar na forma da decomposição resultante e aplicar alguma álgebra de conjuntos. Contudo, não é garantido que as mesmas preservem todas as dependências.

4.6 Conversão para a *BCNF*

O seguinte algoritmo garante a conversão de uma **qualquer relação** R (mesmo sem estar na $2NF$) para a $BCNF$ através de uma decomposição D *lossless*.

Algoritmo 2 Algoritmo de conversão para a $BCNF$

$D \leftarrow \{R\}$
while $\exists S \in D$ not in BCNF **do**
  $Q \leftarrow \{S \in D : S \text{ not in BCNF}\}$
  $(X \rightarrow Y) \leftarrow \{(X \rightarrow Y) \in Q : X \rightarrow Y \text{ violates BCNF}\}$
  $D \leftarrow (D \setminus Q) \cup \{(Q \setminus Y), (X \cup Y)\}$
end while

A última instrução do bloco *while* corresponde a uma decomposição *lossless* pelo teorema de *Heath*: $Q \leftarrow \{(Q \setminus Y), (X \cup Y)\}$. A propriedade da sucessão de decomposições *lossless* juntamente com o facto de que cada iteração diminui em pelo menos um atributo duas relações de D e que, no limite, relações com apenas 2 atributos não geram quaisquer redundância, garante que o algoritmo converte a relação para a $BCNF$ em tempo $O(2^{|R|})$.

Nota: Existe um algoritmo que gera uma decomposição tanto lossless como preservadora de dependências para conversão para a 3NF

Capítulo 5

Índices

Quando se pretende procurar registos numa tabela, uma abordagem passaria por ler toda a tabela e todos os seus conteúdos. Contudo, tal seria demasiado ineficiente para *queries* que retornam poucos resultados.
Desta forma, recorrem-se a **índices**, que são ficheiro auxiliares que, dada uma **chave de procura** (conjunto de atributos de uma relação), permitem obter de forma mais rápida informação sobre registos (como o seu endereço em memória ou até o(s) próprio(s) registo(s) - no caso em que a chave de procura não é uma chave primário).
A forma mais rudimentar de índice consiste num ficheiro com entradas dispotas **sequencialmente** e ordenadas pela **chave de procura**. Por forma a localizar uma dada chave de procura, realiza-se procura binária sobre a tabela. Antes de discutir formas mais sofisticadas de índices, apresentam-se conceitos básicos sobre índices. De notar que as diferentes características existem ortogonalmente (i.e., todas as combinações são admissíveis, **excepto quando indicado explicitamente**).

5.1 Conceitos básicos

5.1.1 Índices ordenados e não ordenados

Dependendo de como os registos a que se referem se encontram fisicamente dispostos em memória, os índices podem-se classificar como:

- **Índice ordenado/primário/*clustering***, se os registos se encontram ordenados em memória igualmente por ordem da chave do índice. Normalmente, chaves primárias como chaves de índice dão origem a índices *clustering*, daí a terminologia. Por exemplo, um índice ordenado cuja chave de procura fosse o *id* do instrutor (coluna da esquerda), seria um índice primário, dada a disposição dos registos em memória.

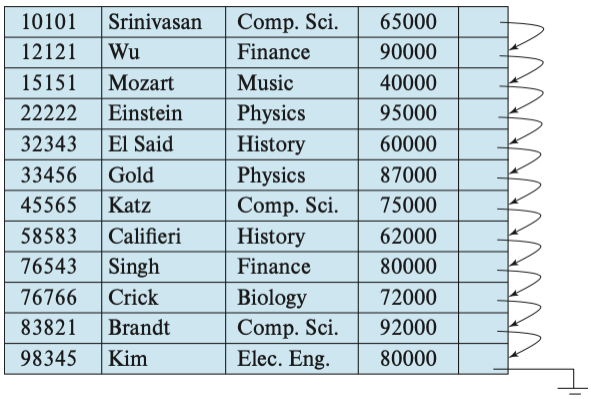


Figura 5.1: Ficheiro com registos da relação *instrutor*

- **Índice não ordenado/secundário/não *clustering***, se a chave do índice especifica uma ordem diferente da ordem sequencial do ficheiro de registos.

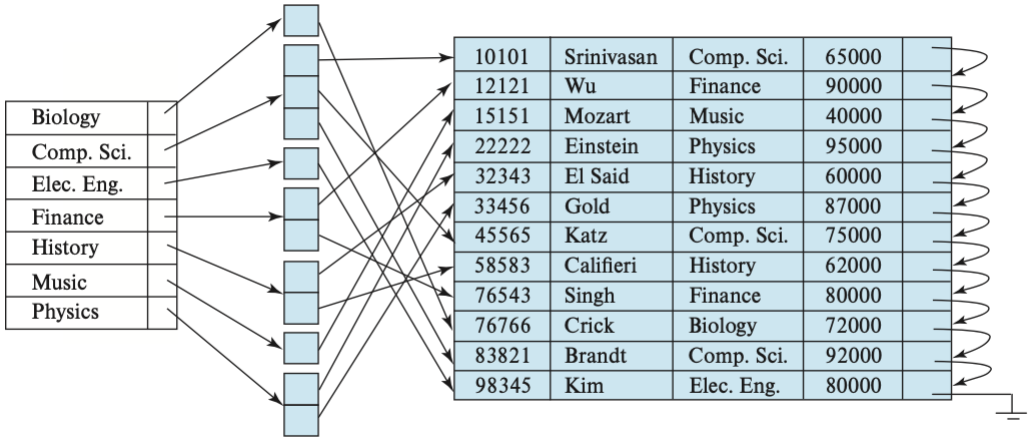


Figura 5.2: Índice secundário sobre o nome do departamento

Índices *clustering* são geralmente **mais vantajosos** que índices não *clustering*, pois, se quisermos obter vários registos com chaves de procura adjacentes, é suficiente ter um ponteiro para o primeiro registo, obtendo os registos da memória sequencialmente avançado simplesmente com esse ponteiro. Nos registos não *clustering*, é possível que o próximo registo esteja num bloco de disco diferente, o que induz maior latência no acesso à memória. Quando se pretende obter **um único registo**, ambos os índices apresentam performances idênticas.

No seguimento desta disussão, é importante referir os tipos de organização física usados pelos SGBD:

- **Heaps**, no qual novos registos são colocados no primeiro espaço livro (leva a fragmentação a longo prazo);
- **Sorted Files**, no qual novos registos os registos são ordenados segundo o valor da sua chave primária;
- **Hashed Files**, no qual os fiheiros são organizados em *buckets* de acordo com uma dada função de *hash*;

5.1.2 Índices Densos/Esparsos

Os índices podem ainda ser:

- **Densos**, se existe uma correspondência 1:1 entre as entrada do índice e registos da tabela.

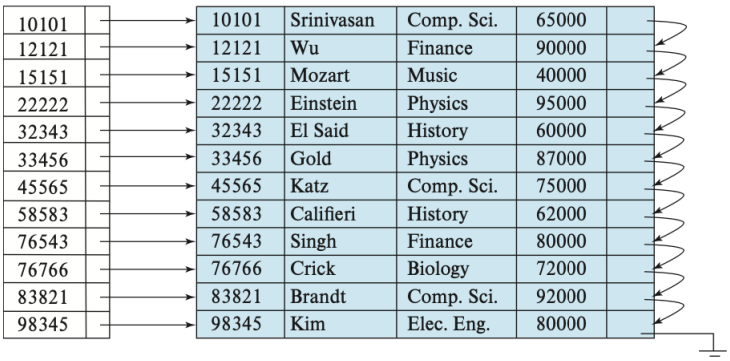


Figura 5.3: Índice denso

- **Esparsos**, se existe uma correspondência 1:N entre uma entrada do índice e registos da tabela, i.e., só existem entradas no índice para alguns registos da tabela - este tipo de organização só é possível em **índices *clustering***. Mais especificamente, para localizar um dado registo, determina-se a entrada no índice com **a maior chave menor ou igual à chave do registo a localizar**. A partir do ponteiro obtido, faz-se um procura sequencial até ao registo pretendido.

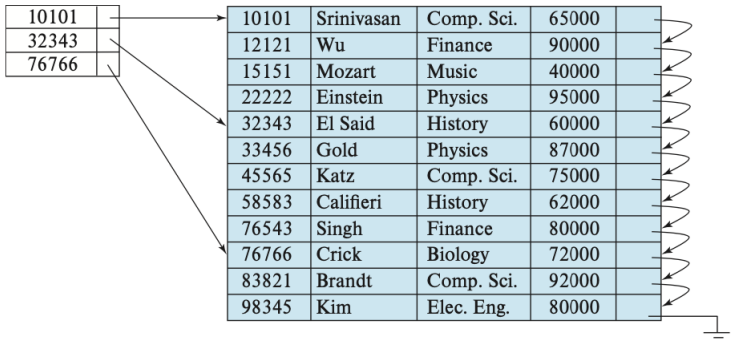


Figura 5.4: Índice esparso

De acordo com as definições anteriores, há um *tradeoff* entre espaço e tempo nas duas variantes anteriores: enquando que um índice esparso ocupa menos espaço em memória, um índice denso permite localizar mais rapidamente registos, ao evitar procura linear sobre as tabelas de registos.
Contudo, atendendo a que o principal *overhead* na execução de uma *query* se prende com a transferência de blocos de e para disco, tem-se que, trazido um bloco para memória, o tempo para pesquisar sobre o bloco é negligenciável. Assim, se cada entrada num índice esparso apontar exatamente para o início de **bloco de índices**, tem-se **índices densos são a opção preferencial**.

5.1.3 Índices Multinível

Se o índice não for pequeno o suficiente para caber em memória principal, a procura de uma entrada no índice pode levar a vários acessos ao disco.
Para lidar com este problema, os **índices multinível** constroem um índice exterior **esparso** sobre o índice original, agora denominado de índice interior (de notar que o índice interior está ordenado por hipótese, permitindo um índice esparso). Cada uma destas tabelas de índices ocupa normalmente **um bloco de disco**, sendo que o índice exterior é pequeno o suficiente por forma a poder ser **mantido em memória principal**.

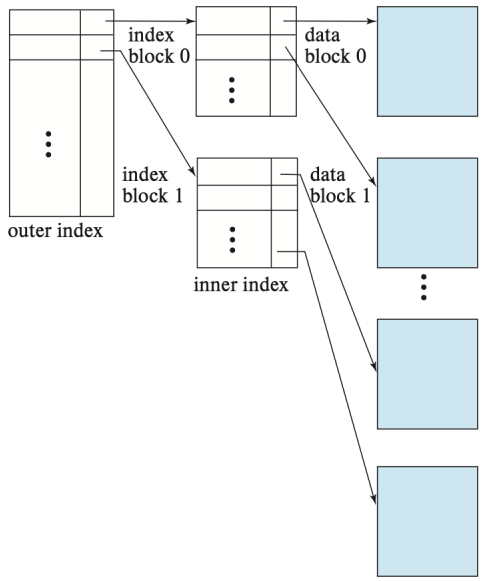


Figura 5.5: Índice Multinível com 2 níveis

Para localizar um índice, faz-se procura binária no índice exterior para encontrar a entrada com **a maior chave menor ou igual que a chave do registo pretendido**. Daqui se obtém um ponteiro para um bloco do índice

interior, que é assim varrido à procura da chave pretendida (se o índice interior for denso) ou da maior chave menor ou igual à pretendida (se o índice interior for esparso).
De notar que é possível adicionar índices interiores sucessivos se a tabela do índice exterior não couber em memória, originando estruturas com mais do que dois níveis.
Assim, o número de leituras de disco é no pior caso N, onde N é o número de níveis do índice multinível - tendencialmente melhor que o limite $\lceil \log_2(b) \rceil$ que se obteria pela pesquisa binária anterior.

5.1.4 Índices em múltiplas chaves

Quando se pretende fazer seleção de registos com base em múltiplos atributos, pode ser conveniente possuir um índice cuja chave é composta por múltiplos atributos.
Desta forma, cada chave do índice é da forma $(a_1, ..., a_n)$ e as entradas do índice encontram-se ordenadas por **ordem lexicográfica** das suas chaves, i.e., $(a_1, b_1) < (a_2, b_2)$ se e só se $a_1 < a_2$ ou $a_1 = a_2$ e $b_1 < b_2$.

5.2 Manutenção de índices

Apesar da aceleração que proporciona, a utilização de índices acarreta custos de manutenação, pois há que manter uma estrutura de índices coerente perante a inserção, remoção e atualização de registos nas tabelas. Como a atualização pode ser vista como um remoção seguida da inserção de um registo diferente do inicial, apenas se abordam os primeiros dois casos.

Inserção

No caso de **índices densos**, insere-se uma entrada no índice com a nova chave. Se o índice for **esparso** e se a novo registo obriga a criação de um novo bloco em disco, então insere-se uma entrada no índice com a nova chave.

Remoção

No caso de **índices densos**, remove-se a entrada no índice correspondente à chave do registo a apagar. Se o índice for **esparso**, há que ter em conta dois casos:

- se a entrada a apagar era a única com a sua chave de procura, então a substitui-se a chave da entrada do índice pela próxima chave; só se apaga a entrada se a próxima chave já possuir uma entrada.
- se a entrada no índice correspondente apontar exatamente para o registo a apagar, então essa entrada é atualizar por forma a apontar para o próximo registo com a mesma chave de procura.

5.3 Tipos de índices

Para além da organização sequencial inicialmente apresentada, existem os seguintes tipos de índices, que apresentam diversas vantagens sobre esta:

5.3.1 Índices $B^+ - tree$

Uma $B^+ - tree$ consiste numa árvora balanceada (i.e., qualquer caminho da raíz até uma folha tem o mesmo comprimento) composta por nós com $\lceil \frac{n}{2} \rceil$ a n filhos, onde n é um parâmetro fixo da árvore (se o nó for uma folha, substitui-se n por $n - 1$ nas condições anteriores).
Cada nó de uma $B^+ - tree$ contém $m - 1$ chaves de procura $K_1, K_2, ..., K_{m-1}$ e m ponteiros $P_1, P_2, ..., P_m$, onde $m \leq n$ e as chaves se encontram por **ordem crescente**.

Figura 5.6: Nó de uma $B^+ - tree$

Nos **nós folha**, cada ponteiro P_i $(i < m)$ aponta para um registo em memória, enquanto o ponteiro P_m aponta para o próximo nó folha, por forma a facilitar varrimentos sequenciais de registos sem ter que percorrer a árvore toda a partir do topo.
Nos **nós interiores**, se possuirem $m-1$ chaves de índice, cada ponteiro P_i $(i \leq m)$ aponta para outro nó que é raíz de uma subárvore com chaves de procura **menores** que K_i e **maiores ou iguais** que K_{i-1}. Notar que o primeiro ponteiro, P_1 aponta para uma subárvore com registos com chaves menores que K_1, e o último ponteiro, P_m, aponta para registos com chaves maiores ou iguais que K_{m-1}.
Se $i > m$ então o ponteiro aponta para **NULL**.

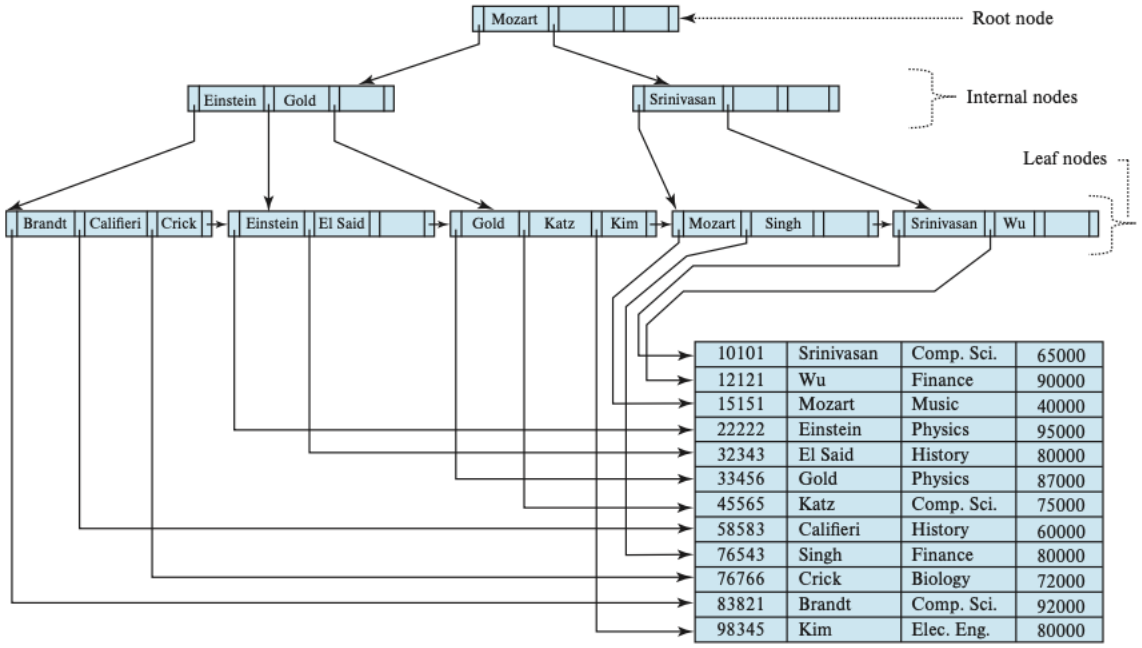


Figura 5.7: Exemplo de uma $B^+ - tree$

Dado que, para uma tabela com N registos, a altura de uma árvore é $O(\log_{\lceil \frac{n}{2} \rceil}(N))$, *queries* de seleção com base em igualdade são muito eficientes quando há índices $B^+ - tree$: basta percorrer uma árvore com altura logarítmica em n desde a raíz até ao nó folha pretendido.
Ademais, ***range queries*** (i.e., procura de valores com chave num intervalo $[\min, \max]$) também são eficientes: basta percorrer a árvore à procura da entrada com chave min e fazer um varrimento sequencial usando os últimos ponteiros de cada nó folha até à última chave com valor max.
Contudo, dada a restrição no número de chaves em cada nó, operações de inserção e remoção induzem um *overhead* adicional, fazendo com que sejam também $O(\log_{\lceil \frac{n}{2} \rceil}(N))$.

5.3.2 Índices *Hash*

Num índice *Hash*, as entradas do índice estão organizadas em *buckets*, que na prática corresponde a uma lista ligada de registos (quando o índice está em memória principal), ou lista ligada de blocos (quando está em disco). Dado um registo com chave K_i, o endereço do *bucket* no qual esse registo vai ser inserido obtém-se por aplicação de uma função de *hash*, $h(K_i)$. Por norma, a resolução de colisões faz-se por encadeamento externo, ou, no contexto de índices, ***overflow chaining***.
Índices *Hash* são particularmente úteis em *queries* de igualdade, não sendo no entando apropriadas para *queries* de *range*.

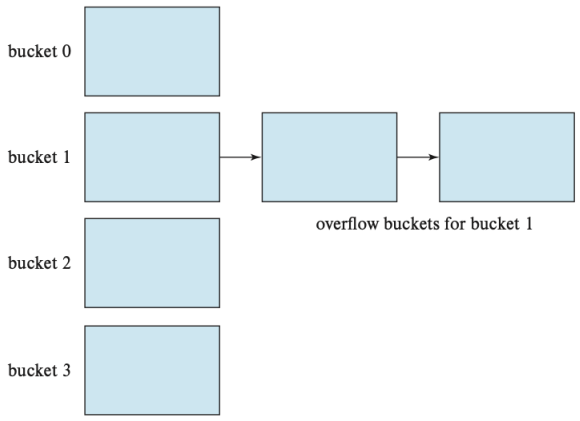


Figura 5.8: Exemplo de um índice *Hash*

Um dos principais problemas associadas a este tipo de índices consiste no *bucket overflow*: quando o número de *buckets* não é suficiente para o número de registos da tabela ou quando certos *buckets* recebem muito mais registos que outros, há que encadear vários *buckets* que na verdade estão associados à mesma chave de dispersão, resultando em piorias na procura de registos.
Por forma a contrariar este problema, em vez de se fazer *hashing* estático (i.e., número de *buckets* é fixo), pode-se fazer *hashing* dinâmico, adaptando-se de forma incremental o número de *buckets* ao número de registos que existem na tabela em cada momento.

5.3.3 Índices *Bitmap*

Um **índice *bitmap*** num atributo A de uma relação r consiste num *bitmap* para cada valor que A pode tomar. Assim, se A tomar valores em $(v_1, ..., v_n)$, existem n bitmaps $b_i = (b_{i1}, ..., b_{iN})$, onde N é o número total de registos na tabela. O significado destes *bitmaps* é o seguinte: se $b_{ij} = 1$, então o registo número j tem o atributo A com valor v_i.

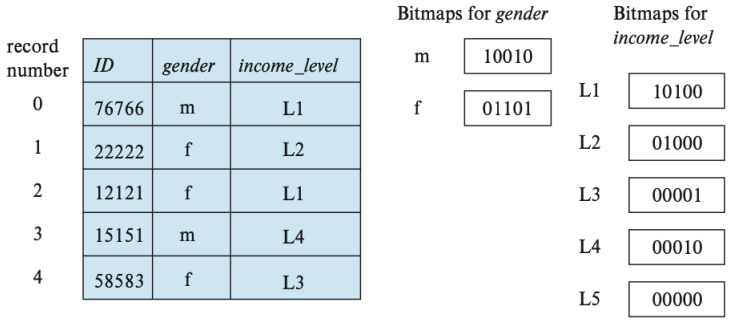


Figura 5.9: Exemplo de um índice *Bitmap*

Os índices *bitmap* revelam-se vantajosos quando as tabelas não são muito grandes e há necessidade de fazer seleções complexas com base em certos atributos. Por exemplo, se se quer determinar os registos com género masculino e com nível de rendimento $L2$ ou $L3$, basta aplicar a expressão lógica aos bitmaps correspondentes. No bitmap resultante, se entrada i está a 1, então o registo 1 satisfaz a *query*, sendo trivial a partir daí saber *exatamente* quais os blocos a aceder sem ter que fazer uma procura sequencial (notar que isto só é vantajoso em *queries* bastante seletivas, como se vai ver mais à frente).

5.4 Índices em SQL

Embora o SQL-92 não defina sintaxe para criação de índices, esta operação é suportada pelos SGBD's atuais:

```
01 |  CREATE INDEX index_name ON table_name [Type] (column list)
```

Onde *Type* corresponde aos tipos de índice anteriores (notar que não é possível fazer um índice *Hash* em mais do que um coluna). A não especificação de *Type* leva a uma escolha *default*.
Por omissão, criam-se índices sobre *primary keys*, dada a necessidade de impor a verificação da restrição da chave primária de forma eficiente. Embora nem sempre se criem índices sobre *foreign keys*, é boa prática fazê-lo, especialmente na presença das cláusulas *ON DELETE CASCADE* ou *ON UPDATE CASCADE*.

5.4.1 Seletividade de uma *query*

Antes de expor em que casos é que é vantajoso e útil criar índices, vamos introduzir o conceito de **seletividade**. Pela discussão anterior, um índice é tão mais útil quanto mais diminuir a necessidade de trazer blocos para memória. Dada uma *query*, a sua **seletividade** está relacionada com a proporção de registos que serão incluídos na resposta (i.e., uma query é **tão mais seletiva** quanto menos registos fizerem parte da resposta). Seja s essa proporção, e seja n o **número de registos que cabem num bloco de disco**. Tem-se:

- A probabilidade de um dado registo não ser incluído na resposta à query é $1 - s$;
- A probabilidade de um bloco não possuir qualquer registo que vai ser incluído na resposta à query é $(1-s)^n$ (assumindo uma distrbuição uniforme dos registos com respostas pelos blocos);
- A probabilidade de um bloco possuir pelo menos um registo que vai ser incluído na resposta à query (sendo por isso trazido para memória) é $1 - (1 - s)^n$.

Atendendo à última expressão, tem-se que um índice serão tão mais útil quando menos registos houver por bloco e quão mais seletiva for a query (i.e., menor s).

5.4.2 *Queries* que dispensam o uso de índices

Queries que não fazem qualquer tipo de *join*, agrupamento, *sorting* ou filtragem dispensam o uso de índices. Exemplo:

```
01 |  SELECT vcName,ntAge FROM tblEmployee1
02 |      UNION ALL
03 |  SELECT vcName,ntAge FROM tblEmployee2
```

Também dispensam o uso de índices **adicionais** *queries* que fazem filtragens/*joins* sobre as chaves primárias de relações (pois já existem índices para estas).

5.4.3 *Queries* onde é conveniente usar índices

Quando há necessidade de remoção de duplicados, há necessidade de ordenar registos, sendo por isso útil criar um índice $B^+ - tree$ sobre os atributos (no caso seguinte, sobre (*vcName*, *ntAge*)).

```
01 |  SELECT vcName,ntAge FROM tblEmployee1
02 |      UNION
03 |  SELECT vcName,ntAge FROM tblEmployee2
```

Quando há seleções com base em igualdade, deve se criar um índice *Hash* sobre o atributo; se houver seleções com base em comparação, esse índice deve ser $B^+ - tree$. Quando existem vários comparações, como heurística, devem ser postos em primeiro lugar no índice os atributos referentes às condições de igualdade (que é uma operação tendencialmente mais seletiva que a comparações de desigualdade).

```
01 |  SELECT * FROM tblEmployee WHERE b=5 AND c>10 AND d=15 AND e<=20
```

Assim, no exemplo acima, qualquer um dos índices é aceitável:

```
01 |  CREATE INDEX idx ON tblEmployee(b,d,c)
02 |  CREATE INDEX idx ON tblEmployee(b,d,e)
03 |  CREATE INDEX idx ON tblEmployee(d,b,c)
04 |  CREATE INDEX idx ON tblEmployee(d,b,e)
```

De qualquer forma, é útil realizar uma **análise prévia da seletividade**, contando, para cada atributo, o número de valores diferentes.

Nota Importante: Apenas é possível criar um índice por tabela!

Também em condições sem prefixo é útil criar índices, pois há na verdade uma comparação alfabética:

```
01 |  SELECT vcName,ntAge FROM tblEmployee WHERE vcName LIKE 'Greg%'
```

5.4.4 *Index-Only Plans*

Se as *queries* referenciarem atributos que são partes de índices, é possível que a query seja respondida apenas com informação presente em índices. Por exemplo, um índice sobre $E(dno, sal)$, permite responder à seguinte *query*:

```
01 |  SELECT E.dno, MIN(E.sal) FROM Emp E
02 |  GROUP BY E.dno
```

5.5 Otimização e Processamento de *queries*

Relembremos as etapas do processamento de uma *query*:

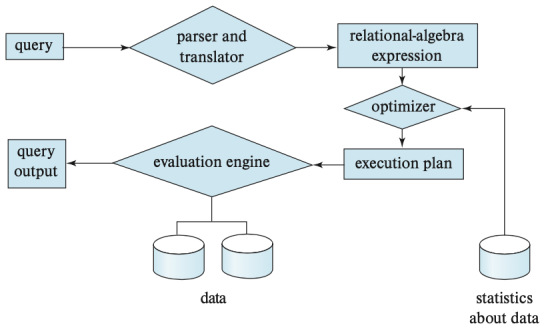


Figura 5.10: Processamento de uma *query*

Por forma a avaliar uma *query*, não é suficiente convertê-la numa expressão de álgebra relacional. É necessário também anotá-la com instruções que especifiquem exatamente como avaliar cada instrução, formando assim um **plano de execução**, que pode ser dado como input ao motor de execução da *query*.
O plano de execução é tendencialmente um plano **ótimo**. Por forma a estimar o custo de cada operação, o SGBD mantém estatísticas sobre cada relação, nomeadamente:

- n_r, o número de tuplos da relação r;
- b_r, o número de bloco que contêm tuplos da relação r;
- l_r, o tamanho de cada tuplo da relação r;
- f_r, o número de tuplos da relação r que cabem num bloco;
- $V(A, r)$, o número de valores distintos que aparecem na relação r no atributo A;

Considera-se ainda o seguinte **modelo de custo**: dado um custo de transferência de um bloco de T_t segundos, um custo de acesso ao bloco de T_s segundos, uma operação que transfira b blocos e que execute S acessos aleatórios (i.e., não sequenciais), terá um custo de

$$b \cdot t_T + S \cdot t_S$$

Vamos agora olhar para os diferentes tipos de operação à luz deste modelo de custo.

5.5.1 Seleções

Seleções sem índice

Na ausência de índice, assumindo contiguidade dos blocos em memória, há que contabilizar o tempo de acesso ao bloco, mais um tempo de transferência por cada bloco trazido para memória. No caso da igualdade na chave, pode-se terminar a pesquisa quando o(s) registo(s) pretendidos são encontrados, o que acontece em média a meio de cada bloco.

Algoritmo	Custo
Pesquisa Linear	$t_S + b_r \cdot t_T$
Pesquisa Linear com igualdade na chave	$t_S + (\frac{b_r}{2}) \cdot t_T$ (caso médio)

Seleções com índice

Estas operações também são frequentemente denominadas de *index scans*.
Seja h_i a altura da árvore $B^+ - tree$ e n o número de registos trazidos para memória. Nos casos abaixo, a igualdade numa chave/não-chave determina se é necessário analisar a totalidade dos blocos apontado por cada entrada no índice. Índices não *clustering* fazem com que o acesso a cada bloco acarrete um custo de t_S adicionais (pois os registos podem se encontrar espalhados na memória). A comparação é semelhante à igualdade numa não-chave.

Algoritmo	Custo
Index *clustering*, índice $B^+ - tree$, igualdade na chave	$(h_i + 1) \cdot (t_T + t_S)$
Index *clustering*, índice $B^+ - tree$, igualdade numa não-chave	$h_i \cdot (t_T + t_S) + t_S + b \cdot t_T$
Index *não clustering*, índice $B^+ - tree$, igualdade na chave	$(h_i + 1) \cdot (t_T + t_S)$
Index não *clustering*, índice $B^+ - tree$, igualdade numa não-chave	$(h_i + n) \cdot (t_T + t_S)$
Index *clustering*, índice $B^+ - tree$, comparação	$h_i \cdot (t_T + t_S) + t_S + b \cdot t_T$
Index não *clustering*, índice $B^+ - tree$, comparação	$(h_i + n) \cdot (t_T + t_S)$

5.5.2 Ordenação

Dada a quantidade de registos a ordenar, raramente a operação de ordenação se realizar totalmente em memória. Dada uma memória na qual cabem M blocos, o **External Sorting** divide os b_r blocos com registos em N *runs* (etapas) com $\lceil \frac{b_r}{M} \rceil$ blocos, ordenando-as individualmente.
De seguida, ocorre um **merge** das N vias (o *mergesort* como o conhecemos fá-lo em 2 vias), diminuindo o número de *runs* em $M - 1$ em cada *pass*.

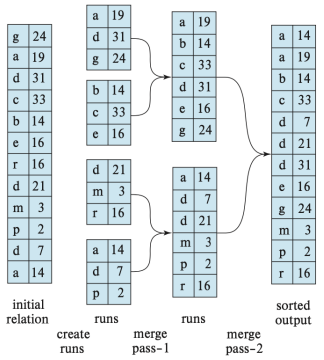


Figura 5.11: *External Sort*

Com certas otimizações, é possível provar que o número de acessos a blocos num *External Sort* é:

$$b_r(2\log_{\lfloor \frac{M}{b_r} \rfloor - 1}(b_r/M) + 1)$$

e o número de acessos aleatórios em disco é (sabendo, por otimização, se trazem b_b de cada run para se fazer merge, em vez de um só bloco):

$$2\lceil b_r/M \rceil + \lceil b_r/b_b \rceil (2\log_{\lfloor \frac{M}{b_r} \rfloor - 1}(b_r/M) - 1)$$

5.5.3 *Joins*

Nested-Loop Join

Dadas relações r e s com n_r e n_s tuplos, este tipo de join apresenta $n_r \cdot b_s + b_r$ acessos a blocos e $n_r + n_s$ acessos aleatórios, sendo a forma mais primitiva de se fazer *join*.

Algoritmo 3 *Nested-Loop Join*

```
for each tuple t_r in r do
    for each tuple t_s in s do
        test pair (t_r, t_s) to see if they satisfy the join condition θ
        if they do, add tr · ts to the result;
    end for
end for
```

Block Nested-Loop Join

Por forma a diminuir a frequência dos acessso aleatórios a disco, este tipo de *join* procede-se bloco a bloco (anteriormente, testava-se um tuplo de r contra *todos* os blocos de s). Este tipo de join apresenta $b_r \cdot b_s + b_r$ acessos a blocos e $b_r + b_s$ acessos aleatórios, o que representa uma melhoria significativa em relação ao caso anterior.

Algoritmo 4 *Block Nested-Loop Join*

```
for each block B_r of r do
    for each block B_s of s do
        for each tuple t_r in B_r do
            for each tuple t_s in B_s do
                test pair (t_r, t_s) to see if they satisfy the join condition θ
                if they do, add tr · ts to the result;
            end for
        end for
    end for
end for
```

Hash Join

Dada uma função de h que mapeia atributos sobre os quais se faz *join* para $\{0, 1, ..., n_h\}$.
Sejam $\{r_0, r_1, ..., r_{n_h}\}$ as partições dos tuplos de r tais que cada tuplo t_r de r é colocado em r_i se:

$$i = h(t_r([\text{Atributos de Join}]))$$

sendo as partições $\{s_0, s_1, ..., s_{n_h}\}$ são definidas analogamente.
O funcionamento do *Hash Join* baseia-se na seguinte observação: se t_r e t_s satisfazem a condição de *join*, então são ambos *hashed* para o mesmo valor i. Desta forma, só é necessário comparar tuplos de r em r_i com tuplos de s em s_i.
Por questões de otimização, para cada i, constroi-se uma tabela $Hash$ com os tuplos de s_i com uma função de

hash diferente da anterior. O obietivo é evitar que todos os registos dentro de um mesmo balde i sejam tidos em conta para fazer *join* com um registo de r.

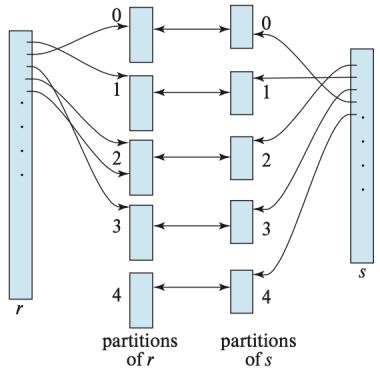


Figura 5.12: *Hash Join*

5.5.4 Otimização de *Queries* baseadas em heurísticas

Por vezes, o número de planos gerados para cada *query* é demasiado grande, originando a um esforço computacional que é agravado pelo cálculo dos custos de cada operação.
Desta forma, dadas de associatividade, comutatividade e distribuição da álgebra relacional que não são aqui referidas, procede-se simplesmente à simplificação da expressão algébrica por forma a que **operações de seleção e de projeção ocorram o mais cedo possível na computação**.
O *rational* desta decisão prende-se com a diminuição da quantidade de informação a processar o mais cedo possível, descartando colunas e registos das tabelas que não serão usados. (Note-se que nem sempre isso ocorre: se existir um seleção depois de um join e houver um índice sobre atributos do join, pode ser prejudicial adiantar a operação de seleção).

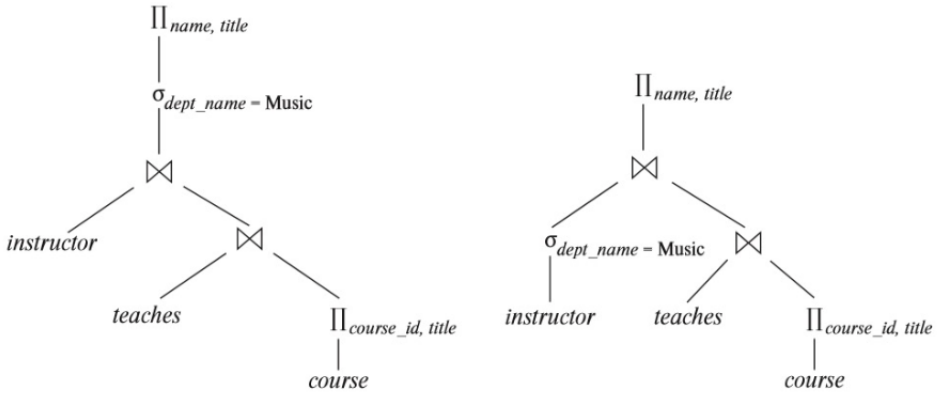


Figura 5.13: Otimização da expressão algébrica por heurísticas

5.5.5 Análise de planos de execução no Postgres

Através do comando:

```
01 |  EXPLAIN ANALYSE
02 |  [query]
```

é possível obter uma **árvore correspondente ao plano de execução**. Se for pretendido obter **dados reais** relativos à execução da *query*, pode-se envolter o comando numa transação:

```
01 | BEGIN TRANSACTION
02 | EXPLAIN ANALYSE
03 | [query]
04 | ROLLBACK
```

```
Sort  (cost=717.34..717.59 rows=101 width=488) (actual time=7.761..7.774 rows=100 loops=1)
  Sort Key: t1.fivethous
  Sort Method: quicksort  Memory: 77kB
  ->  Hash Join  (cost=230.47..713.98 rows=101 width=488) (actual time=0.711..7.427 rows=100 loops=1)
        Hash Cond: (t2.unique2 = t1.unique2)
        ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244) (actual time=0.007..2.583 rows=10000 loops=1)
        ->  Hash  (cost=229.20..229.20 rows=101 width=244) (actual time=0.659..0.659 rows=100 loops=1)
              Buckets: 1024  Batches: 1  Memory Usage: 28kB
              ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101 width=244) (actual time=0.080..0.526 rows=100 loops=1)
                    Recheck Cond: (unique1 < 100)
                    ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0) (actual time=0.049..0.049 rows=100 loops=1)
                          Index Cond: (unique1 < 100)
Planning time: 0.194 ms
Execution time: 8.008 ms
```

Figura 5.14: Resultado do comando *EXPLAIN ANALYSE*

```
Sort
└── Hash Join
    ├── Seq Scan
    └── Hash
        └── Bitmap Heap Scan
            └── Bitmap Index Scan
```

Figura 5.15: Árvore de execução correspondente

O termo *cost* diz respeito ao custo estimado da operação, sendo *actual time* o custo real; *width* diz respeito ao tamanho de cada tuplo, *rows* ao número de registos analisados e *loops* o número de vezes em que se iterou sobre um dado conjunto de registos. Descreve-se agora o significado de alguns dos procedimentos que se podem obter:

- **SeqScan**: um varrimento sequencial da tabela;
- **IndexScan**: varrer o índice, ir de seguida à memória.
- **IndexOnlyScan**: varrer unicamente o índice (acontece em *queries* que manipulam/acedem a atributos de um dado índice);
- **Bitmap Scan**: varrer o índice e construir um *bitmap* dos registos que obedecem ao critério de filtragem.
- **NestLoop/HashJoin**: falados anteriormente;
- **Merge Join**: tipo de merge que se faz em tempo linear quando ambos os conjuntos de tuplos estão ordenados;
- **Lateral Join**: *Join* contra *table expressions*;
- **Semi Join**: para *outer joins*;
- **Anti Join**: *joins* exclusivos (e.g., NOT IN...);
- **GroupAggregarte**: agregação normal;
- **HashAggregarte**: agregação em memória baseada em funções de *hash*;
- **CTEScan**: junção de uma CTE à *query* principal;
- **SubqueryScan**: idem, mas para *subqueries*;
- **Materialize**: criar um conjunto de registos em memória a partir de uma *query*;
- **Append**: usado na junção de conjuntos de registos.

Nota: a existência de SeqScan's pode evidenciar a necessidade de um índice.

Nota: uma discrepância entre valores previstos/reais pode ser resolvida através do comando VACUUM ANALYSE, que força o recalculo das estatísticas do SGBD.

Capítulo 6

Transações

6.1 Introdução às Transações

Definição 15 (Transação)**.** Uma transação é um conjunto de operações de um programa que formam uma unidade lógica de trabalho na qual podem ser acedidos e atualizados vários dados.

Existem 2 questões a resolver nas transações:

- **Concorrência**: a execução concorrente de várias transações - resolvida com tem múltiplos processadores disponíveis para múltiplos utilizadores simultâneos;
- **Integridade**: lidar com falhas de vários tipos, nomeadamente de *hardware*, *crashes* do sistema operativo e falhas de *software* do SGBD - resolvida pelas garantidas de integridade do próprio SGBD.

Existem, como tal, 3 caminhos possíveis para a conclusão de uma transação:

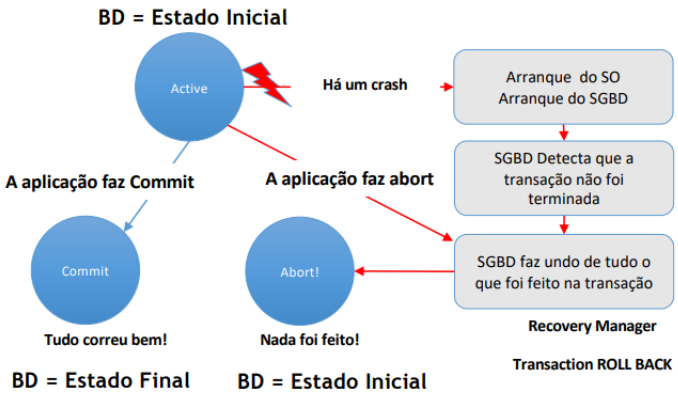


Figura 6.1: 3 caminhos possíveis de uma transação

As transações possuem 4 grandes propriedades (**ACID**):

- **Atomicidade**: numa transação, as alterações ao estado são **atómicas**: ou todas se realizam ou nenhuma se realiza - a função do sistema é manter informação sobre as alterações efetuadas por cada transação ativa e, em caso de *crash* ou *abort* explícito, desfazer as alterações feitas desde o início da transação até ao ponto de rutura.
- **Consistência**: uma transação é uma **transformação correta** do estado, por exemplo, o conjunto das ações da transação não viola nenhuma das regras de integridade associadas ao estado - a função do sistema é assegurar que a base de dados evolui de um estado coerente para outro estado coerente. Os estados coerentes são definidos pela lógica aplicacional.
- **Isolamento**: embora as transações se executem concorrentemente, os estados intermédios de uma transação são invisíveis a todas as restantes transações. Estas vêm apenas ou o estado inicial ou o estado final - a

função do sistema é garantir que uma transação apenas "vê"(leituras/escritas) alterações realizadas por transações *committed*.

– **Durabilidade**: uma vez completada uma transação (*commit* concluído), todas as alterações ao estado são imutáveis, sobrevivendo a qualquer falta do sistema - a função do sistema é manter informação sobre alterações efetuadas por cada uma das transações *committed* e, em caso de *crash*m refazer as alterações que ainda não se encontravam registadas em disco.

Podemos esquematizar uma transação com o seguinte diagrama de estados:

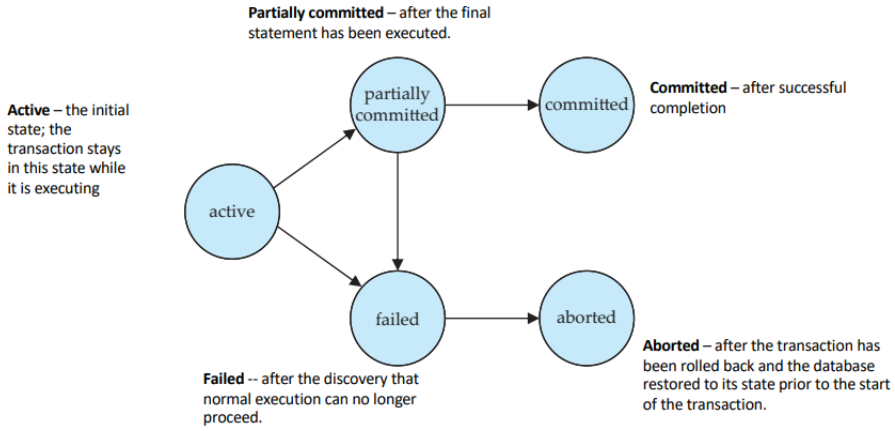


Figura 6.2: Diagrama de estados de uma transação

6.2 Transações em SQL

Uma **transação** em SQL consiste na sequência de instruções de consulta e/ou atualização. O SQL *standard* especifica que uma transação começa implicitamente quando uma instrução SQL é executada.
Uma das seguintes instruções SQL deve terminar a transação:

– **COMMIT [WORK]** confirma a transação ativa; i.e, faz as atualizações realizadas pela transação permanentes na base de dados.

– **ROLLBACK [WORK]** causa a transação ativa a ser desfeita; i.e, desfaz todas as atualizações feitas por instruções SQL na transação, pelo que a base de dados volta ao estado imediatamente antes da transação.

1) Seja a seguinte transação: "Escreva uma transação que permita transferir 350€ da conta A para a conta B.". Temos:

```
01 |  -- Verificar saldos
02 |  SELECT balance
03 |  FROM account
04 |  WHERE account_number='A';
05 |
06 |  SELECT balance
07 |  FROM account
08 |  WHERE account_number='B';
09 |
10 |  -- Transferir 350 euros de A para B
11 |  START TRANSACTION;
12 |  -- Retirar de A
13 |  UPDATE account
14 |  SET balance = balance - 350
15 |  WHERE account_number='A';
16 |  -- Adicionar a B
17 |  UPDATE account
18 |  SET balance = balance + 350
19 |  WHERE account_number='B';
20 |  -- Confirmar a transacao
21 |  COMMIT;
```

Vários sistemas usam *auto-commit* por defeito, onde o início explícito de início de transação é omitido, e cada consulta é uma transação - se houver erros dá **ROLLBACK** automático, c.c, **COMMIT** automático.
Para lidar com a concorrência, normalmente usam-se modelos de trincos, e trancam-se tuplos envolvidos numa operação antes de lhes aceder. Para a seguinte consulta que tenciona ver em que departamento é que os instrutores chamados "João Aragonez"trabalham:

```
01 |  SELECT dept_name
02 |  FROM instructor
03 |  WHERE name = 'Joao Aragonez';
```

seria necessário bloquear toda a relação *instructor*, para assegurar que não possam ser inseridos novos registos com *name* = 'João Aragonez'. Porém, trancar a relação inteira implica acabar com concorrência.
Para combater este problema, existem níveis de isolamento menos exigente, onde algumas operações não exigem 100% de consistência, por exemplo, o saldo médio de todas as contas registadas num banco, o cálculo de dados estatísticos para otimização de operações. A solução passa por um *trade-off* entre exatidão dos resultados e desempenho do sistema, preferindo que neste tipo de transações, não seja feito a seralização com outras, ou seja, poupam-se as verificações e deixa-se a transação correr livremente em paralelo.
Eis os níveis de consistência em SQL:

- **Serializable**: por defeito.
- **Repeatable read**: relativamente igual à **Serializable**, mas permite por exemplo uma transação T_1 fazer uma consulta sobre o número de instrutores chamados João Aragonez e haver outra transação T_2 que cria ou modifica um tuplo contendo um instrutor chamado João Aragonez antes que T_1 seja confirmado.
- **Read committed**: só permite a leitura de tuplos confirmados;
- **Read uncommitted**: qualquer tuplo não confirmado pode ser lido.

Para alterarmos o nível de consistência devemos usar o seguinte comando:

```
01 |  SET TRANSACTION ISOLATION LEVEL
02 |      { SERIALIZABLE
03 |      | REPEATABLE READ
04 |      | READ COMMITTED
05 |      | READ UNCOMMITTED
06 |      }
```

Definimos ainda sobre os níveis de isolamento em SQL:

- **phantom read**: fazendo a mesma consulta duas vezes, o número de registos pode ser diferente, se entretanto outra transação que inseriu registos foi confirmada.
- **nonrepeatable read**: fazendo a mesma consulta duas vezes, cada registo pode conter dados diferentes, se entretanto outra transação que fez **UPDATE** foi confirmada.
- **dirty read**: fazendo a mesma consulta duas vezes, é possível ver os dados alterados por outras transações que estão a correr e ainda nem sequer foram confirmadas.

Nível de isolamento	*dirty reads*	*non-repeatable reads*	*phantom reads*
SERIALIZABLE	não	não	não
REPEATABLE READ	não	não	possível
READ COMMITTED	não	possível	possível
READ UNCOMMITTED	possível	possível	possível

Figura 6.3: Consistência e Isolamento em SQL

6.3 Concorrência

Logo no 1º capítulo vimos que um SGBD contém um módulo *Concurrency Manager*, que por sua vez garante **C**oerência e **I**solamento das transações.

6.3.1 Isolamento de Transação e Serialização

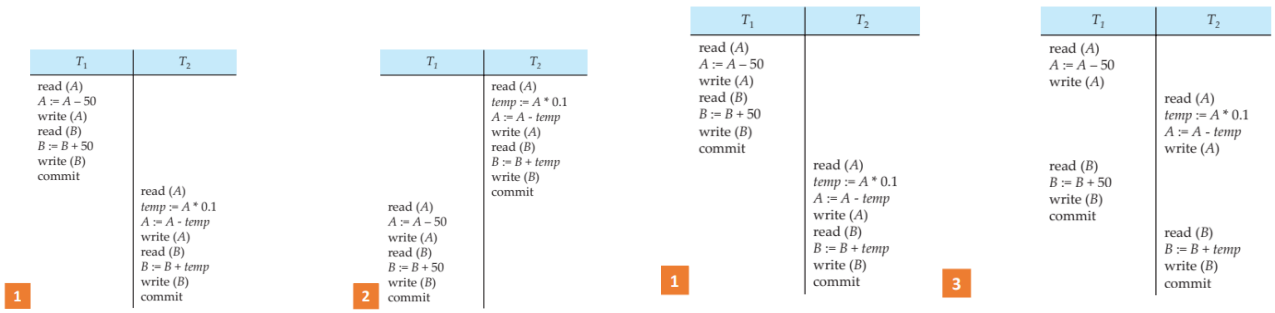
A execução de transações concorrentes **em série** é ineficiente, pelo que se permite que várias transações corram simultaneamente no sistema. Isto traz algumas **vantagens**:

- **Maior utilização do disco e processadores** - aumento do *throughput*;
- **Redução do tempo médio de resposta** das transações.

Definição 16 (Escalonamento/*Schedule*)**.** Sequência de instruções que especificam a ordem cronológica na qual as instruções de transações concorrentes são executadas.

- Um escalonamento de um conjunto de transações inclui todas as instruções de todas as transações.
- **Tem de preservar a ordem** na qual as instroções aparecem em cada transação individual.

Um escalonamento diz-se **serial** se na sequência de instruções das várias transações, as instruções relativas a uma única transação aparecem seguidas.



(a) Escalonamentos em Série

(b) Escalonamentos Equivalentes

Figura 6.4: Escalonamento de Transações

Se repararmos nos escalonamentos equivalentes, notamos que o escalonamento não precisa ser serial para produzir o resultado correto, mas tem de ser **equivalente** a um escalonamento serial para o considerarmos **serializável**. Para os escalonamentos introduzimos uma **visão simplificada** das transações, considerando apenas as instruções de leitura e escrita na análise destas.

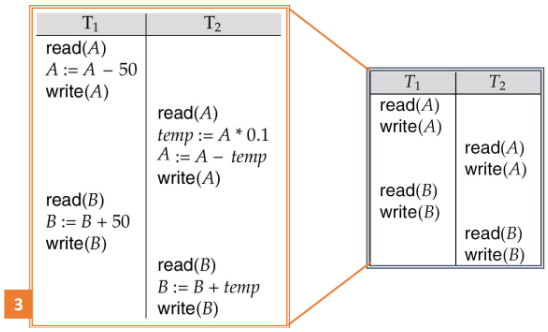


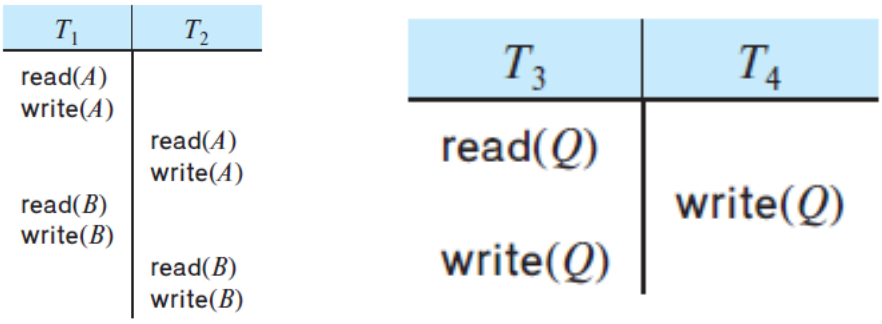
Figura 6.5: Visão simplificada de uma Transação

Consideremos agora um escalonamento S, no qual existem 2 instruções consecutivas, I e J, das transações T_i e T_j respetivamente ($i \neq j$). Se I e J se referirem a objetos diferentes, então podemos trocar I e J sem afetar os resultados de qualquer instrução do escalonamento. Contudo, se se refirem ao mesmo objeto Q, então a ordem

destes passos é relevante. Dada a **visão simplificada** das transações, só consideramos as instruções de leitura e escrita e temos os seguintes 4 casos:

- I: **read**(Q), J: **read**(Q) - a ordem de I e J é irrelevante, visto que o mesmo valor de Q vai ser lido por T_i e T_j, independentemente da ordem;
- I: **read**(Q), J: **write**(Q) - se I vem antes de J, então T_i não lê o valor de Q que está a ser escrito por T_j na instrução J. Se J vier antes de I, então T_i lê o valor a ser escirot. Logo, a ordem é relevante.
- I: **write**(Q), J: **read**(Q) - análogo ao caso anterior.
- I: **write**(Q), J: **write**(Q) - visto que ambas as instruções são de escrita, a ordem não afeta nenhuma das transações T_i e T_j. Contudo, o valor a ser obtido na próxima instrução de **read(Q)** de S é afteada, visto que o resultado de apenas 1 das escritas é preservado na base de dados. Assim, a ordem é também relevante.

Dizemos então que I e J têm um **conflito** se são operações de **transações diferentes** que acedem ao mesmo objeto, e pelo menos 1 dessas instruções é uma **escrita**.
Se I e J são instruções de transações diferentes e não têm conflito entre si, então podemos trocar a ordem para criar um novo escalonamento **S'**, que é equivalente a **S**, visto que todas as instruções estão na mesma ordem exceto para I e J, cuja ordem é irrelevante - se um escalonamento **S** pode ser transformado noutro escalonamento **S'** por uma série de trocas de instruções não-conflituosas, diz-se que **S** e **S'** são **equivalentes sem conflitos**. Dizemos também que um escalonamento **S** tem **conflitos serializáveis** se é equivalente a um escalonamento serial.



(a) Escalonamento com Conflito Serializável

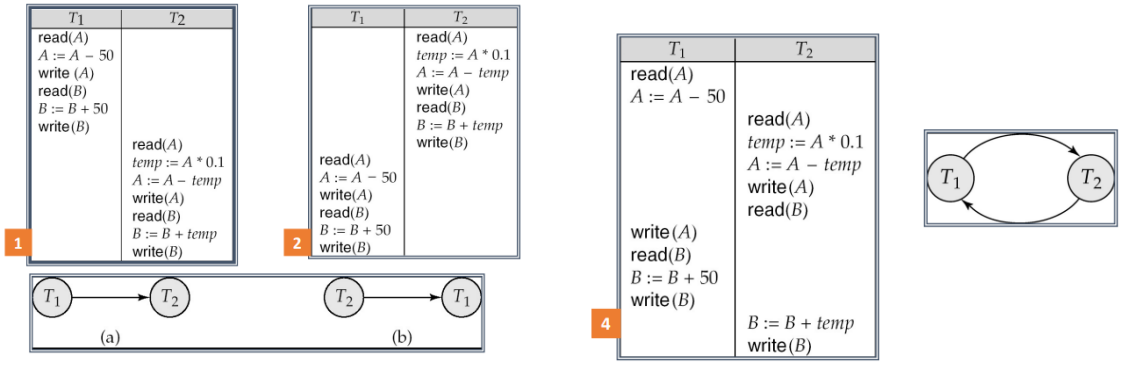
(b) Escalonamento com Conflito Não Serializável

Figura 6.6: Escalonamento com Conflitos

De notar que a figura em (b) tem um conflito não serializável, pois se tivéssemos $< T_3, T_4 >$, o próximo **read**(Q) iria ler o resultado da transação T_4, e se tivermos $< T_4, T_3 >$ a transação T_3 irá ler o valor da escrita em T_4.
Uma maneira simples e eficiente de determinar conflitos de serialização num escalonamento é construir um **grafo de precedência** a partir de um escalonamento **S**. O grafo consiste num par $G = (V, E)$, onde V é um conjunto de vértices e E um conjunto de arestas. O conjunto de **vértices** V é dado por todas as transações T_i que participam no escalonamento. O conjunto de **arestas** consiste em todas as arestas $T_i \rightarrow T_j$ no qual 1 das 3 condições se verificam:

- T_i executa **write**(Q) antes de T_j executar **read**(Q);
- T_i executa **read**(Q) antes de T_j executar **write**(Q);
- T_i executa **write**(Q) antes de T_j executar **write**(Q);

Se uma aresta $T_i \rightarrow T_j$ existe no grafo de precedência, então em qualquer escalonamento serial **S'** equivalente a **S**, T_i tem de aparecer **antes** de T_j.



(a) Escalonamento Serializável (b) Escalonamento Não Serializável

Figura 6.7: Escalonamento Serializável

Um escalonamento é **serializável** se o seu grafo de precedências não tiver ciclos - se o grafo for acíciclo, podemos obter uma **ordenação topológica** e executar as transações sobre uma qualquer. De notar que isto é uma condição **suficiente**, mas não **necessária**, pois existem casos de escalonamentos serializáveis cujo grafo de precedência contém um ciclo.

6.3.2 Isolamento e Atomicidade de uma Transação

Até agora vimos escalonamentos que assumem que não existem falhas nas transações.
Se uma transação T_i falha, precisamos reverter os seus efeitos, e os efeitos de qualquer transação T_j que esteja a ser executada concorrentemente com T_i e que dependa desta.
Um **escalonamento parcial** define-se por **não ter** as operações de **commit** ou **abort** em todas as transações que este abrange, como é o caso do escalonamento abaixo.

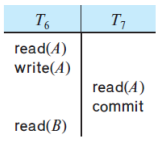


Figura 6.8: Escalonamento não recuperável

Como T_7 confirma a transação logo a seguir a ler A, i.e., ainda quando T_6 está ativa. Se T_6 falhar antes de confirmar, terá de reverter todas as suas instruções e T_7 já confirmou uma transação de um valor que fora revertido - T_7 é **dependente** de T_6, pelo que devemos abortar também T_7 por questões de atomicidade. Como tal, diz-se que este escalonamento é **não-recuperável**.

Definição 17 (Escalonamento Recuperável)**.** Um escalonamento diz-se recuperável sse para cada par de transações T_i e T_j, tal que T_j lê dados escritos previamente por T_i, então a confirmação de T_i tem de aparecer antes da confirmação de T_j.

Para o exemplo acima ser recuperável, T_7 teria de esperar até T_6 confirmar para poder confirmar a sua própria transação.
Isto pode criar o efeito de **rollbacks em cadeia**, como é o seguinte caso:

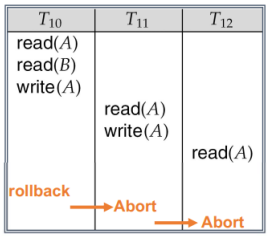


Figura 6.9: Rollbacks em cadeia

Uma pequena extensão que se fazem às transações são **savepoints** - são usados dentro das transações como "pontos seguros"das aplicações cujo estado é conhecido e a partir dos quais a computação pode recomeçar. Os **rollbacks** podem ser para **savepoints**.

6.3.3 Protocolo de Isolamento com Trincos

Uma maneira de assegurar isolamento é requisitar que objetos de dados sejam acedidos por exclusão mútua, i.e, se uma transação acede um objeto, nenhuma outra transação pode modificar esse mesmo - para isso usamos **trincos**.
Existem 2 modos de trincos sobre objetos:

- **Partilhado**. Se uma transação T_i obteve um **trinco partilhado** (seja este **S**) no objeto Q, então T_i pode ler, mas não pode escrever em Q.
- **Exclusivo**. Se uma transação T_i obteve um **trinco exclusivo** (seja este **X**) no objeto Q, então T_i pode ler e escrever em Q.

A transação **pede** o modo de trinco apropriado de acordo o tipo de operações que vai executar sobre o objeto Q - o pedido é feito ao *Concurrency Control Manager* do SGBD, e bloqueia até lhe ser concedido (fica em *wait-state*).

	S	X
S	true	false
X	false	false

Figura 6.10: Matriz de Compatibilidade dos Modos de Trinco

De forma simples, o protocolo de acesso a objetos nas transações concorrentes é dado por:

- Uma transação tem que adquirir um trinco **partilhado** antes de tentar ler o valor de um tuplo.
- Uma transação tem que adquiri um trinco **exclusivo** antes de tentar escrever o valor de um tuplo.
- Os trincos são **libertados** aquando da confirmação da transação.

6.3.4 Protocolo Two-Phase Locking

O **protocolo two-phase locking** assegure serialização. Requer que cada transação emita pedidos de bloqueio e desbloquio de trinco em 2 fases:

- Fase **Crescente**/*Growing Phase*: uma transação pode obter trincos, mas não pode libertá-los;
- Fase **Decrescente**/*Shrinking Phase*: uma transação pode libertar trincos, mas não pode obter nenhum trinco novo;

Inicialmente uma transação está na fase crescente e adquire trincos à medida que necessita. Mal liberte um dos trincos, entra na fase decrescente e não pode emitir mais pedidos.

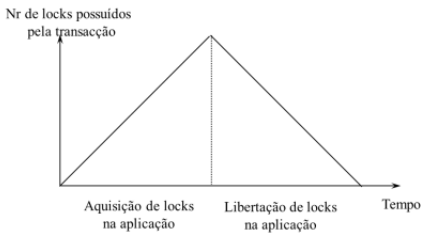


Figura 6.11: Gráfico de Trincos vs Tempo no 2PL

Rollbacks por cascata podem ser evitado usando uma modificação do *2PL* chamada **Protocolo Strict Two-Phase Locking** - igual à 2PL, porém os trincos **exclusivos** ficam trancados até a transação ter sido confirmada.

Outra variante é o **Protocolo Rigorous Two-Phase Locking** - igual à S2PL, mas aplica-se também aos trincos partilhados e não só aos exclusivos.
Por fim à concorrência, existe também uma política de controlo de concorrência alternativa aos trincos - **timestamps**. Se a transação T_1 iniciou a execução antes de T_2, então o sistema deve garantir o resultado da serialização $\{T_1, T_2\}$, oue seja, T_1 não pode ler dados alterados por T_2 e T_1 não pode alterar dados que T_2 já tenha lido.

6.4 Recuperação

Uma parte integral de um SGBD é um **esquema de recuperação** que consegue restaurar a base de dados para um estado consistente que existia antes da falha.

6.4.1 Classificação de Falha

Existem duas categorias de falhas:

- **Falhas de sistema** - afeta as transações no sistema, sem haver deficiência permanente nele. (ex.: falta de corrente elétrica, erro de software);
- **Falhas de componentes** - existe uma avaria num dos componentes do sistema. (ex.: avaria do disco rígido).

6.4.2 Armazenamento

Quando o SGBD arranca após uma falta tem que recuprar as transações em curso, analisando um **registo de log** e refazer as transações com confirmação aceite e desfazer as que não têm - só depois é que o sistema está operacional.
Existem 3 tipos de armazenamento:

- Armazenamento volátil: não sobrevive ao *crash* e *boot* do sistema;
- Armazenamento não-volátil: sobrevive ao *crash* e *boot* do sistema;
- **Armazenamento estável**: tem a redundância de *hardware* para permitir N falhas de *hardware*.

O **armazenamento estável** desempenha um papel crítico em algoritmos de recuperação.