

# Apontamentos de BD

João Aragonez

# Conteúdo

<b>1</b>	<b>Conceitos Iniciais</b>	<b>3</b>
1.1	Sistemas de informação . . . . .	3
1.2	Sistemas de Gestão de Bases de Dados (SGBD) . . . . .	3
1.2.1	Vantagens dos SGBD's . . . . .	4
1.2.2	Desvantagens dos SGBD's . . . . .	4
1.3	Modelos e Níveis de Abstração nos SI . . . . .	4
1.4	Modelos de Dados . . . . .	5
1.5	Arquitetura dos SGBD . . . . .	5
1.6	Conceção de Bases de Dados . . . . .	6
1.7	Utilizadores de Bases de Dados . . . . .	6
<b>2</b>	<b>SQL</b>	<b>7</b>
2.1	Visão Global da Linguagem <i>Query</i> SQL . . . . .	7
2.2	Definição de Dados em SQL . . . . .	7
2.2.1	Tipos Básicos . . . . .	7
2.2.2	Definição Básica de Esquemas . . . . .	8
2.3	Estrutura Básica de Consultas SQL . . . . .	8
2.3.1	Consultas em 1 Relação . . . . .	9
2.3.2	Consultas em N Relações . . . . .	10
2.4	Operações Básicas Adicionais . . . . .	11
2.4.1	Operação de Renomeação . . . . .	11
2.4.2	Operações em Cadeias de Caracteres . . . . .	12
2.4.3	Especificação de Atributo na cláusula SELECT . . . . .	13
2.4.4	Ordenação da Disposição dos Tuplos . . . . .	13
2.4.5	Predicados da cláusula WHERE . . . . .	13
2.4.6	Operações de Conjuntos . . . . .	14
2.5	Valores NULL . . . . .	14
2.6	Funções de Agregação . . . . .	15
2.6.1	Agregação Básica . . . . .	15
2.6.2	Agregação com Agrupamento . . . . .	16
2.6.3	Cláusula HAVING . . . . .	16
2.6.4	Agregação com Valores NULL e Booleanos . . . . .	16
2.7	Sub-Consultas Aninhadas . . . . .	17
2.7.1	Pertença de Conjuntos . . . . .	17
2.7.2	Comparação de Conjuntos . . . . .	17
2.7.3	Teste para Relações Vazias . . . . .	18
2.7.4	Teste para Ausência de Tuplos Duplicados . . . . .	18
2.7.5	Sub-Consultas na cláusula FROM . . . . .	19
2.7.6	Cláusula WITH . . . . .	19
2.7.7	Sub-Consultas Escalares . . . . .	19
2.8	Modificação da Base de Dados . . . . .	20
2.8.1	Remoção . . . . .	20
2.8.2	Inserção . . . . .	20
2.8.3	Atualizações . . . . .	20
2.9	Expressões JOIN . . . . .	21

2.9.1	Natural JOIN . . . . .	21
2.9.2	Condições JOIN . . . . .	22
2.9.3	Outer JOIN . . . . .	22
2.9.4	Tipos e Condições de JOIN . . . . .	24
2.10	Vistas . . . . .	24
2.10.1	Definição de Vista . . . . .	25
2.10.2	Vistas em Consultas SQL . . . . .	25
2.10.3	Vistas Materializadas . . . . .	25
2.10.4	Atualização de Vistas . . . . .	26
<b>3</b>	<b>Transações</b>	<b>27</b>
3.1	Introdução às Transações . . . . .	27
3.2	Transações em SQL . . . . .	28

# 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 iterrações (*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.

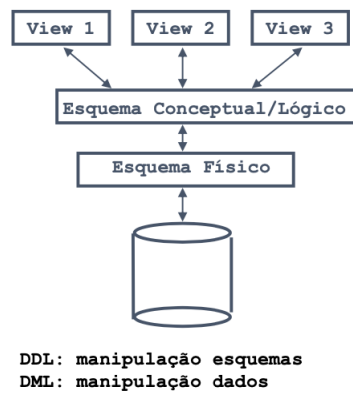
### 1.2.2 Desvantagens dos SGBD's

- **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/armazenamento:

- **Modelo Conceptual** (ou esquema externo), que descreve como os utilizadores vêem os dados. Permite particularizar 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).

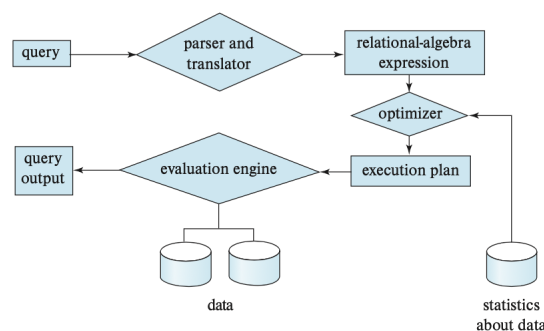
Entre outros modelos de dados, destacam-se o **Modelo Relacional**, o **Modelo Entidade-Associação**, o **Modelo Baseado em Objetos**, **Modelos de Dados Semi-Estruturados** (como *XML/JSON*), ou os **Modelos em Rede e Hierárquicos** (não usados atualmente).

Contudo, o modelo de dados mais amplamente difundido nos SGBD 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 exhibe 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**.

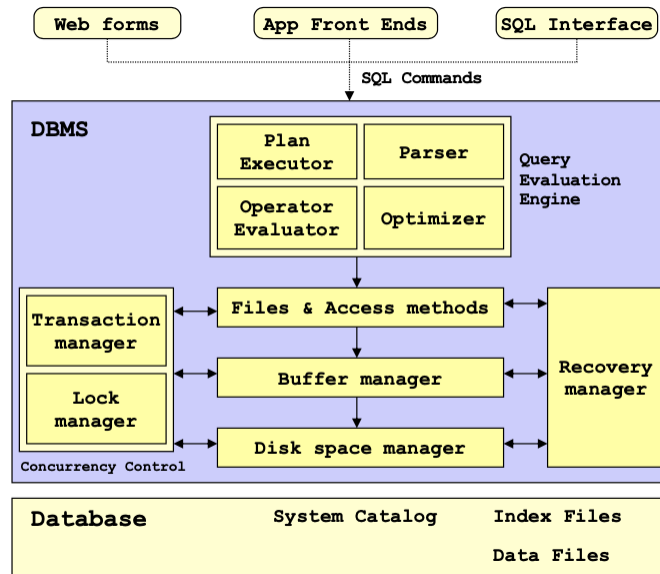


Figura 1.3: Arquitetura de um SGBD

## 1.6 Conceção de Bases de Dados

O processo de concepção de um base de dados incide inicialmente no **desenho lógico**, i.e., sobre o **esquema** a adotar. Para esta 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:

```
01 | CREATE TABLE departamento
02 |     (nome_dept VARCHAR(20),
03 |     edificio VARCHAR(15),
04 |     orcamento NUMERIC(12, 2)),
05 |     PRIMARY KEY (nome_dept));
```

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

```
01 | CREATE TABLE r
02 |     (A_1 D_1,
03 |     A_2 D_2,
04 |     ...,
05 |     A_n D_n,
06 |     <restricao-integridade 1>,
07 |     ...,
08 |     <restricao-integridade k>);
```

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

- **primary key** ( $A_{j1}, A_{j2}, \dots, A_{jm}$ ): esta especificação diz que os atributos  $A_{j1}, A_{j2}, \dots, A_{jm}$  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** ( $A_{k1}, A_{k2}, \dots, A_{kn}$ ) **references s**: esta especificação diz que os valores dos atributos ( $A_{k1}, A_{k2}, \dots, A_{kn}$ ) 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:

```

01 | CREATE TABLE department
02 |     (dept name VARCHAR (20),
03 |     building VARCHAR (15),
04 |     budget NUMERIC (12,2),
05 |     PRIMARY KEY (dept name));
06 | CREATE TABLE course
07 |     (course id VARCHAR (7),
08 |     title VARCHAR (50),
09 |     dept name VARCHAR (20),
10 |     credits NUMERIC (2,0),
11 |     PRIMARY KEY (course id),
12 |     FOREIGN KEY (dept name) REFERENCES department);
13 | CREATE TABLE instructor
14 |     (ID VARCHAR (5),
15 |     name VARCHAR (20) NOT NULL,
16 |     dept name VARCHAR (20),
17 |     salary NUMERIC (8,2),
18 |     PRIMARY KEY (ID),
19 |     FOREIGN KEY (dept name) REFERENCES department);
20 | CREATE TABLE section
21 |     (course id VARCHAR (8),
22 |     sec id VARCHAR (8),
23 |     semester VARCHAR (6),
24 |     year NUMERIC (4,0),
25 |     building VARCHAR (15),
26 |     room number VARCHAR (7),
27 |     time slot id VARCHAR (4),
28 |     PRIMARY KEY (course id, sec id, semester, year),
29 |     FOREIGN KEY (course id) REFERENCES course);
30 | CREATE TABLE teaches
31 |     (ID VARCHAR (5),
32 |     course id VARCHAR (8),
33 |     sec id VARCHAR (8),
34 |     semester VARCHAR (6),
35 |     year NUMERIC (4,0),
36 |     PRIMARY KEY (ID, course id, sec id, semester, year),
37 |     FOREIGN KEY (course id, sec id, semester, year) REFERENCES section,
38 |     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**.

```

01 | SELECT name
02 | 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:

```

01 | SELECT DISTINCT dept_name
02 | 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:

```

01 | SELECT ID, name, dept_name, salary * 1.1
02 | FROM instructor;

```

4) Seja a seguinte consulta: "Encontra os nomes 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 Inform 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 presente 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 cláusula **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 cláusula **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.

<i>instructor.ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>	<i>teaches.ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>
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 especificados 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 alternativos 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 = 'Física'
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  $(v_1, v_2, \dots, v_n)$  para denotar um tuplo de aridade  $n$  contendo os valores  $v_1, v_2, \dots, v_n$  - 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*:

```

01 | SELECT name, course_id
02 | FROM instructor, teaches
03 | 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 1º semestre do ano de 2018.". Podemos decompor isto:

```

01 | -- O conjunto de cursos lecionados no 2 semestre de 2017
02 | SELECT course_id
03 | FROM section
04 | WHERE semester=2 AND year=2017;
05 |
06 | -- O conjunto de cursos lecionados no 1 semestre de 2018
07 | SELECT course_id
08 | FROM section
09 | WHERE semester=1 AND year=2018;
10 |
11 | -- A uniao destes da-nos o resultado da consulta (sem duplicados)
12 | (SELECT course_id
13 | FROM section
14 | WHERE semester=2 AND year=2017)
15 | UNION
16 | (SELECT course_id
17 | FROM section
18 | WHERE semester=1 AND year=2018);
19 |
20 | -- A uniao destes da-nos o resultado da consulta (com duplicados)
21 | (SELECT course_id
22 | FROM section
23 | WHERE semester=2 AND year=2017)
24 | UNION ALL
25 | (SELECT course_id
26 | FROM section
27 | 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 1<sup>o</sup> 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 na 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.** 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 1º semestre 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 1º semestre 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 j   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 Cláusula 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 department)
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 retorne 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:

```

01 | SELECT dept_name,
02 |       (SELECT COUNT(*)
03 |        FROM instructor
04 |        WHERE department.dept_name=instructor.dept_name) AS num_instructors
05 | FROM department;

```

## 2.8 Modificação da Base de Dados

Vejam 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:

```

01 | DELETE FROM r
02 | 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:

```

01 | DELETE FROM instructor
02 | WHERE dept_name IN (SELECT dept_name
03 |                   FROM department
04 |                   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:

```

01 | INSERT INTO course
02 | 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:

```

01 | INSERT INTO instructor
02 | SELECT ID, name, dept_name, 18000
03 | FROM student
04 | 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:

```

01 | UPDATE instructor
02 | SET salary = salary * 1.05
03 |
04 | -- 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, descubra 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 **JOINED** 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 (simétrico)
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**, estende 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  $\cup$  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 especificar 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	Join conditions
inner join	natural
left outer join	on <predicate>
right outer join	using ( $A_1, A_2, \dots, A_n$ )
full outer join	

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 personalizada de relações "virtuais"<sup>1</sup> 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
```

<sup>1</sup>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.

```
06 |      AND section.year = 2017;
```

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 cláusula **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.

# Capítulo 3

## Transações

### 3.1 Introdução às Transações

**Definição 7.** 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:

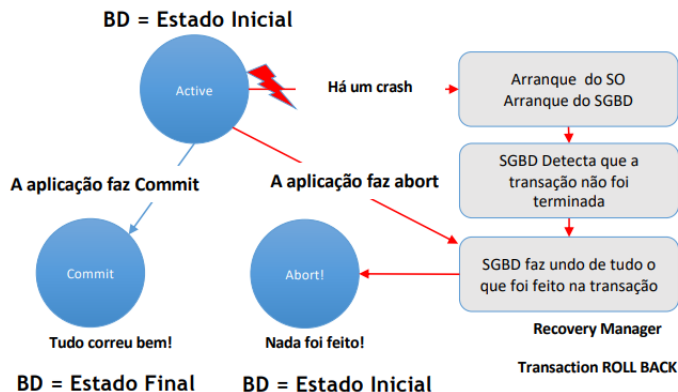


Figura 3.1: 3 caminhos possíveis de uma transação

As transações tem 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* refazer as alterações que ainda não se encontravam registadas em disco.

Podemos esquematizar uma transação com o seguinte diagrama de estados:

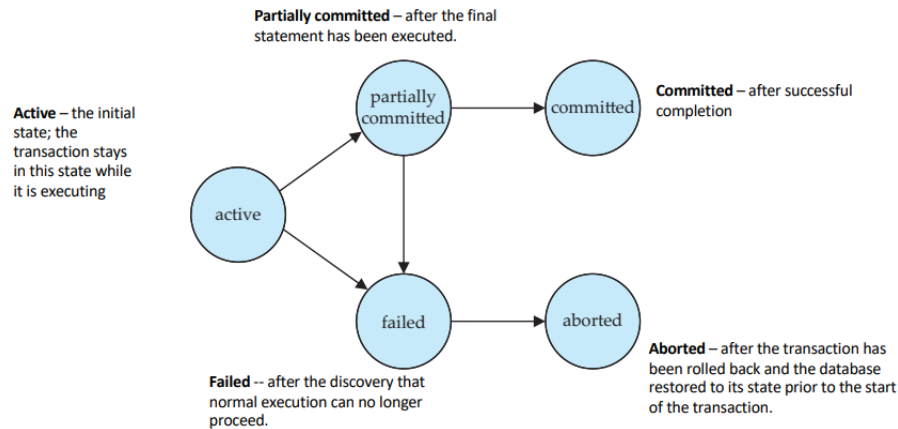


Figura 3.2: Diagrama de estados de uma transação

## 3.2 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 transação
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 | { READ UNCOMMITTED
03 |   | READ COMMITTED
04 |   | REPEATABLE READ
05 |   | SERIALIZABLE
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 entre-tanto 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	<i>dirty reads</i>	<i>non-repeatable reads</i>	<i>phantom reads</i>
<b>SERIALIZABLE</b>	não	não	não
<b>REPEATABLE READ</b>	não	não	possível
<b>READ COMMITTED</b>	não	possível	possível
<b>READ UNCOMMITTED</b>	possível	possível	possível

Figura 3.3: Consistência e Isolamento em SQL