Universidade do Minho

Ano Letivo: 2022/23
Turnos: PL1/PL8

Bases de Dados

PL10/11 – Exploração de Bases de Dados e SQL Avançada

Docente: Diana Ferreira
Email: diana.ferreira@algoritmi.uminho.pt

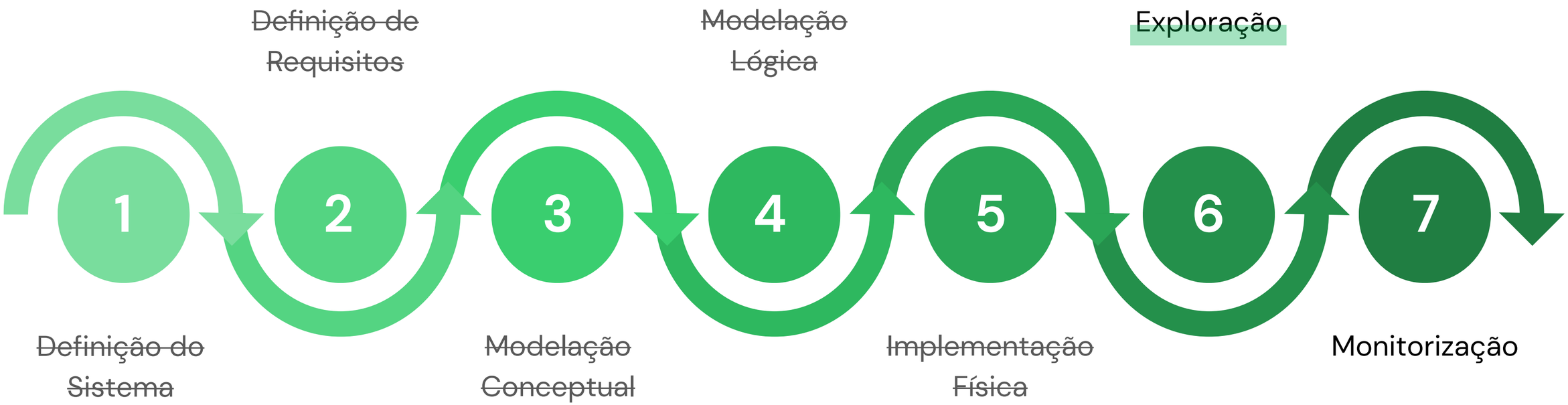
Horário de Atendimento:
4ª feira 18h-19h

Sumário

Bibliografia:

- Connolly, T., Begg, C., Database Systems, A Practical Approach to Design, Implementation, and Management , Addison-Wesley, 4a Edição, 2004. **(Chapter 8)**
- Belo, O., "Bases de Dados Relacionais: Implementação com MySQL", FCA – Editora de Informática, 376p, Set 2021. ISBN: 978-972-722-921-5. **(Capítulo 5, 6 e 7)**

Ciclo de vida de um SBD



FASE 6: Exploração

➡ Data Manipulation Language (DML)

Existem 4 instruções básicas para a manipulação de dados:

- **INSERT** → para inserir dados na BD;

```
INSERT INTO <nome_tabela> (<c1>,<c2>,...) VALUES (<v1>,<v2>,...);
INSERT INTO <nome_tabela> (<c1>,<c2>,...)
VALUES
  (<v11>,<v12>,...),
  ...
  (<vnn>,<vn2>,...);
```

- **DELETE** → para remover dados da BD;

```
DELETE FROM <nome_tabela> WHERE <condição>;
```

- **SELECT** → para consultar dados da BD;

```
SELECT [DISTINCT] {* | <nome_c1>, ...}
FROM <nome_tabela>,...
[WHERE <condição>]
[ORDER BY <c1> [ASC | DESC], ...];
```

- **UPDATE** → para atualizar dados da BD;

```
UPDATE <nome_tabela>
SET
   <c1> = <v1>,
   <c2> = <v2>,
   ...
[WHERE <condição>];
```

FASE 6: Exploração

Operações de Junção

A operação de Junção é utilizada para combinação dos dados contidos numa ou mais tabelas através das colunas em comum, ou seja, as *foreign keys*. A cláusula JOIN é usada na instrução SELECT e aparece sempre depois da cláusula FROM.

O mysql suporta diferentes operações de junção:

- CROSS JOIN;
- NATURAL JOIN;
- INNER JOIN;
- LEFT JOIN;
- RIGHT JOIN;

FASE 6: Exploração

CROSS JOIN

O produto cartesiano, ou CROSS JOIN, é uma operação entre duas relações R e S que dá origem a uma relação que é a concatenação de cada tuplo de R relacionada com cada tuplo de S, ou seja, combina cada linha da tabela R com cada linha da tabela S. O esquema da relação resultante contém todas as colunas de R e de S, apresentadas pela ordem com que aparecem respetivamente em R e em S.

R

a
b

S

1
2
3

=

RxS

a	1
a	2
a	3
b	1
b	2
B	3

$$SELECT * FROM R CROSS JOIN S;$$
$$SELECT * FROM R, S;$$

FASE 6: Exploração

CROSS JOIN

EXEMPLOS:

- Combinação da relação “medicos” com a relação “especialidades” retorna:

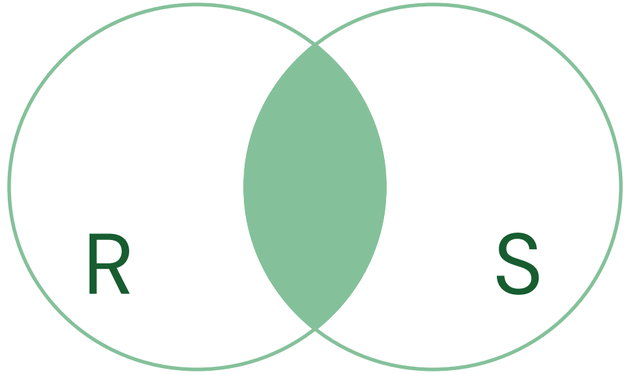
$$SELECT\ *\ FROM\ medicos\ CROSS\ JOIN\ especialidades;$$

num_mec	cod_especialidade	cod_especialidade	des_especialidade
24	2221	2200	Anestesiologia
24	2221	2201	Angiologia
24	2221	2202	Cardiologia
24	2221	2203	Cirurgia Cardiotorácica
24	2221	2204	Cirurgia Geral
24	2221	2205	Cirurgia Vascular
24	2221	2206	Dermatologia
24	2221	2207	Endocrinologia
24	2221	2208	Estomatologia
24	2221	2209	Fisiatria
24	2221	2210	Gastrenterologia
24	2221	2211	Ginecologia
24	2221	2212	Medicina Interna
…	…	…	…

FASE 6: Exploração

NATURAL JOIN

A operação de Junção Natural, é uma operação entre duas relações R e S que permite inter-relacionar essas duas relações através das colunas que sejam comuns às duas relações e que possuam valores iguais. O esquema da relação resultante contém todas as colunas de ambas as relações – excluindo-se uma das colunas de junção.



$$SELECT\ *\ FROM\ R\ NATURAL\ JOIN\ S;$$

NOTA: Se as duas relações envolvidas numa operação de junção natural não possuírem qualquer coluna em comum, então a operação de junção natural é equivalente a um produto cartesiano entre as duas relações.

FASE 6: Exploração

NATURAL JOIN

EXEMPLOS:

- Quais são as especialidades exercidas pelos médicos?

$$SELECT\ *\ FROM\ especialidades\ NATURAL\ JOIN\ medicos;$$

- Liste os pacientes com seguro de saúde.

$$SELECT\ *\ FROM\ pacientes\ NATURAL\ JOIN\ seguros\ WHERE\ dta\_fim\ >\ curdate();$$

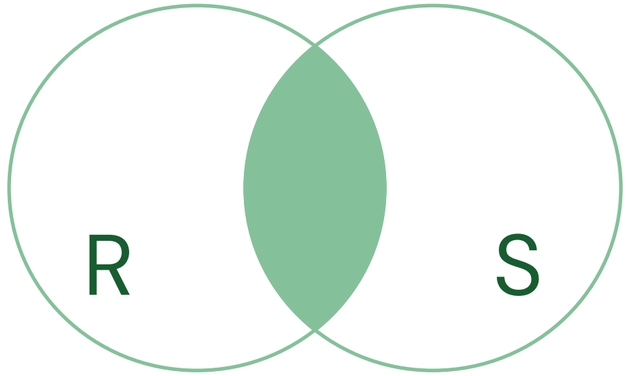
- Liste para cada prescricao, o nome do farmaco, a quantidade prescrita e a unidade.

$$SELECT\ m.nome, p.quantidade, p.unidade\ FROM\ prescricoes\ p\ NATURAL\ JOIN\ medicamentos\ m;$$

FASE 6: Exploração

INNER JOIN

A operação de Junção Interna, é uma operação entre duas relações R e S que permite inter-relacionar essas duas relações através das colunas que satisfaçam a expressão predicativa. O esquema da relação resultante contém todas as colunas de ambas as relações.



$$SELECT * FROM\ R\ INNER\ JOIN\ S\ ON\ R.A = S.B;$$

Para além do operador de igualdade (=), podem ser usados os operadores >, < e <>.

$$SELECT * FROM\ R\ INNER\ JOIN\ S\ USING\ (A);$$

Se as colunas de junçao das duas tabelas tiverem o mesmo nome.

FASE 6: Exploração

INNER JOIN

EXEMPLOS:

- Quais são as especialidades exercidas pelos médicos?

$$SELECT\ *\ FROM\ especialidades\ INNER\ JOIN\ medicos\ USING(cod\_especialidade);$$

- Quais são os medicos que deram consultas?

$$SELECT\ *\ FROM\ medicos\ m\ INNER\ JOIN\ consultas\ c\ ON\ m.nr\_mec = c.nr\_mec\_medico;$$

- Quais as consultas cujo custo final foi inferior ao custo da consulta por especialidade?

$$SELECT\ *\ FROM\ consultas\ c$$
$$INNER\ JOIN\ medicos\ m\ ON\ c.nr_{mec_{medico}} = m.nr_{mec}$$
$$INNER\ JOIN\ especialidades\ e\ ON\ e.cod\_especialidade = m.cod\_especialidade\ AND\ e.preco\_consulta > c.custo\_final;$$

FASE 6: Exploração

LEFT JOIN

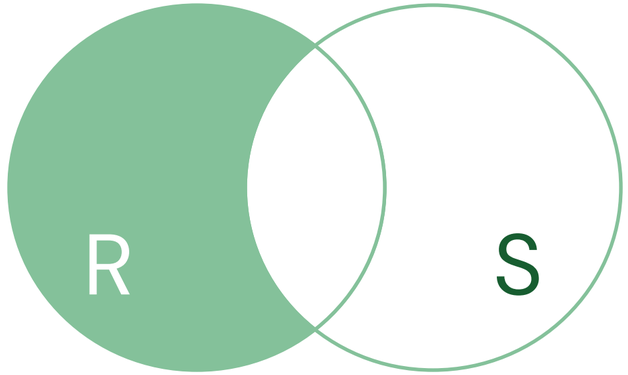
A operação de Junção Externa à Esquerda (*Outer Left Join*), integra na relação final todas as tuplas da relação à esquerda, mesmo quando estas não obedecem aos critérios de junção definidos. Ou seja, os tuplos de R que não têm correspondência nas colunas comuns de S são incluídos no resultado. Quando não existem valores correspondentes na segunda relação S, apresentam-se valores nulos (NULL).

$$SELECT\ *\ FROM\ R\ LEFT\ JOIN\ S\ ON\ R.A = S.B;$$
$$SELECT\ *\ FROM\ R\ LEFT\ JOIN\ S\ USING(A);$$

FASE 6: Exploração

LEFT JOIN

Como a operação de Junção Externa à Esquerda (*Outer Left Join*) integra na relação final todas as tuplas da relação à esquerda R, mesmo quando não têm correspondência na relação à direita S (ou seja apresentam-se valores nulos), é possível selecionar apenas as tuplas da relação R que não têm correspondência na relação S usando a cláusula WHERE e o operador IS NULL.



$$SELECT * FROM R LEFT JOIN S ON R.A = S.B WHERE S.ID IS NULL;$$
$$SELECT * FROM R LEFT JOIN S USING(A) WHERE S.ID IS NULL;$$

FASE 6: Exploração

LEFT JOIN

EXEMPLOS:

- Quais os nomes dos médicos que nunca deram consultas?

$$SELECT\ f.nome\ FROM\ funcionarios\ f\ NATURAL\ JOIN\ medicos\ m$$
$$\textbf{LEFT JOIN}\ consultas\ c\ ON\ c.nr\_mec\_medico = m.nr\_mec\ WHERE\ c.nr\_episodio\ IS\ NULL;$$

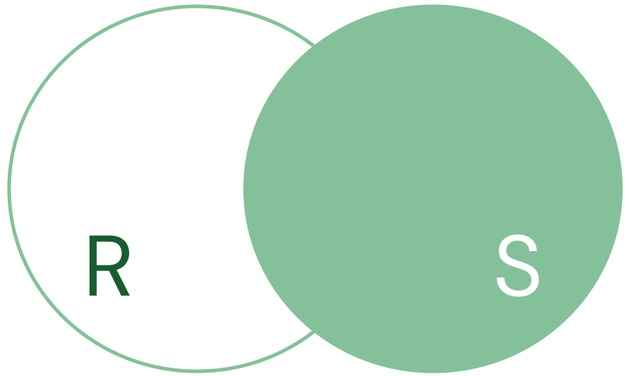
- Quais os nomes dos medicamentos que nunca foram prescritos?

$$SELECT\ m.nome\ FROM\ medicamentos\ m$$
$$\textbf{LEFT JOIN}\ prescricoes\ p\ USING\ (id\_med)$$
$$WHERE\ p.id\_med\ IS\ NULL\ AND\ p.nr\_episodio\ IS\ NULL;$$

FASE 6: Exploração

RIGHT JOIN

A operação de Junção Externa à Direita (*Outer Right Join*), é semelhante Junção Externa à Esquerda, exceto que o tratamento das tabelas unidas é invertido. Ou seja, integra na relação final todas as tuplas da relação à direita, mesmo quando estas não obedecem aos critérios de junção definidos. Quando não existem valores correspondentes na primeira relação R, apresentam-se valores nulos (NULL).

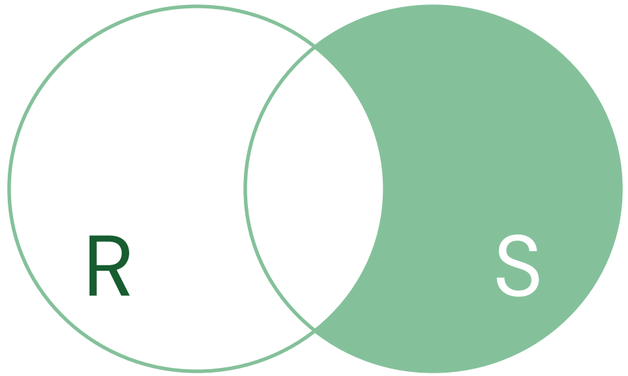


$$SELECT\ *\ FROM\ R\ RIGHT\ JOIN\ S\ ON\ R.A = S.B;$$
$$SELECT\ *\ FROM\ R\ RIGHT\ JOIN\ S\ USING(A);$$

FASE 6: Exploração

RIGHT JOIN

Como a operação de Junção Externa à Direira (*Outer Right Join*) integra na relação final todas as tuplas da relação à direita S, mesmo quando não têm correspondência na relação à esquerda R (ou seja apresentam-se valores nulos), é possível selecionar apenas as tuplas da relação S que não têm correspondência na relação R usando a cláusula WHERE e o operador IS NULL.



$$SELECT * FROM R RIGHT JOIN S ON R.A = S.B WHERE R.ID IS NULL;$$
$$SELECT * FROM R RIGHT JOIN S USING(A) WHERE R.ID IS NULL;$$

FASE 6: Exploração

RIGHT JOIN

EXEMPLOS:

- Liste todos os seguros e os respectivos nomes dos pacientes;

$$SELECT\ nome, nr\_apolice\ FROM\ seguros\ \textbf{\textit{LEFT JOIN}}\ pacientes\ USING\ (nr\_apolice); SELECT\ nome, nr$$

- Quais os nomes dos médicos que nunca deram consultas?

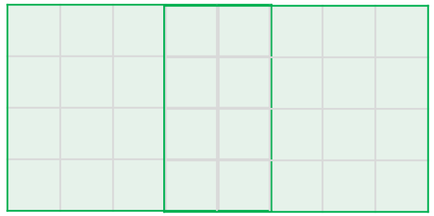
$$SELECT\ f.nome\ FROM\ consultas\ c\ \textbf{\textit{RIGHT JOIN}}\ medicos\ m\ NATURAL\ JOIN\ funcionarios\ f$$
$$ON\ c.nr\_mec\_medico = m.nr\_mec\ WHERE\ c.nr\_episodio\ IS\ NULL;$$

$$SELECT\ f.nome\ FROM\ funcionarios\ f\ NATURAL\ JOIN\ medicos\ m$$
$$\textbf{\textit{LEFT JOIN}}\ consultas\ c\ ON\ c.nr\_mec\_medico = m.nr\_mec\ WHERE\ c.nr\_episodio\ IS\ NULL;$$

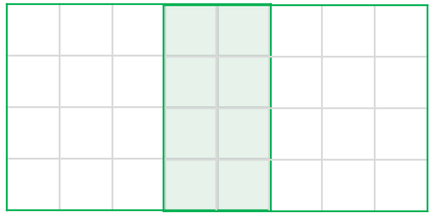
FASE 6: Exploração

Operações de conjuntos

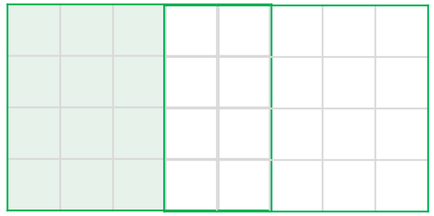
As operações sobre conjuntos unem duas relações, eliminando tuplas repetidas da relação.



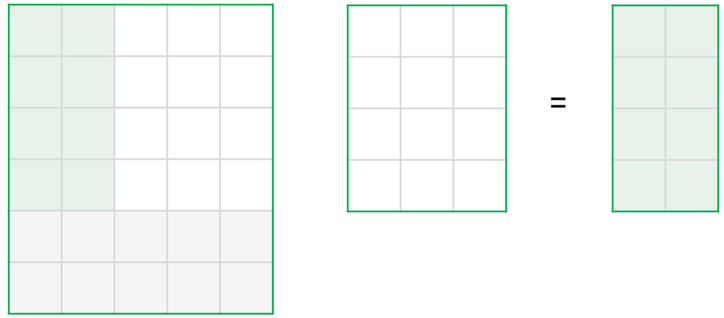
$R \cup S$ - união



$R \cap S$ - intersecção



$R - S$ - diferença

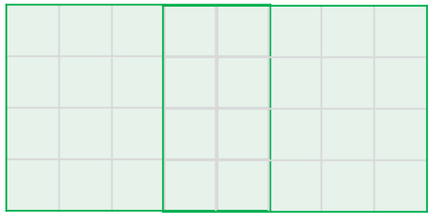


$R \div S$ - divisão/quociente

FASE 6: Exploração

Operação de União

A operação de União é uma operação entre duas relações compatíveis R e S que gera uma relação que contém todas as tuplas pertencentes a R, a S, ou a ambas, eliminando tuplas repetidas. Diz-se que duas relações são compatíveis se possuírem o mesmo grau (nº de colunas) e se as colunas correspondentes forem do mesmo domínio (tipo de dados).



união

```
SELECT * FROM R
UNION (ALL)
SELECT * FROM S;
```

*** Usa-se o UNION ALL quando queremos incluir linhas repetidas.**

NOTA: Na qual as relações R e S possuem o mesmo nº de colunas e com domínios equivalentes.

FASE 6: Exploração

Operação de União

EXEMPLOS:

- Liste os emails, tanto dos pacientes como dos funcionários, numa única relação.

$$SELECT\ *\ FROM\ telefones\_pacientes\ UNION\ SELECT\ *\ FROM\ telefones\_func;$$

- Liste os nomes dos pacientes de Braga ou dos pacientes que foram consultados ou ambos.

Pacientes de Braga

$$SELECT\ nome\ FROM\ pacientes\ WHERE\ localidade =' Braga';$$

Pacientes que foram consultados

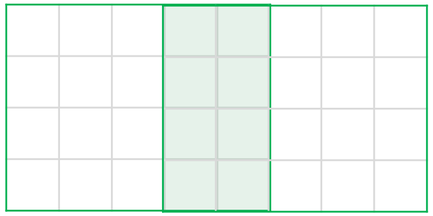
$$SELECT\ nome\ FROM\ pacientes\ NATURAL\ JOIN\ consultas;$$

$$SELECT\ nome\ FROM\ pacientes\ WHERE\ localidade = 'Braga'$$
$$UNION\ SELECT\ nome\ FROM\ pacientes\ NATURAL\ JOIN\ consultas;$$

FASE 6: Exploração

Operação de Intersecção

A operação de Intersecção é uma operação entre duas relações compatíveis R e S que gera uma relação com esquema igual a R que contém todas as tuplas que pertencem simultaneamente a R e a S. Diz-se que duas relações são compatíveis se possuírem o mesmo grau (nº de colunas) e se as colunas correspondentes forem do mesmo domínio (tipo de dados).



interseção

$$SELECT * FROM R$$
$$INTERSECT$$
$$SELECT * FROM S;$$

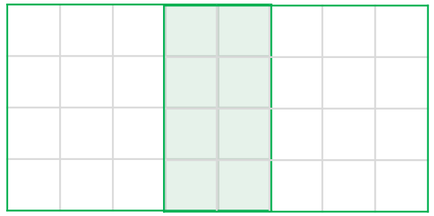
O INTERSECT não é suportado pelo MySQL.

No entanto, esta operação pode ser representada usando outras operações como exists, in, any, e join.

NOTA: Na qual as relações R e S são compatíveis.

FASE 6: Exploração

Operação de Intersecção



interseção

```
SELECT * FROM R
INTERSECT
SELECT * FROM S;
```

O INTERSECT não é suportado pelo MySQL.

```
SELECT * FROM R
WHERE r1 IN
(SELECT r1 FROM S);
```

```
SELECT * FROM R
WHERE EXISTS
(SELECT * FROM S
WHERE R.r1 = S.r1);
```

```
SELECT * FROM R
WHERE r1 = ANY
(SELECT r1 FROM S );
```

```
SELECT * FROM R
INNER JOIN S USING
(r1);
```

NOTA: Na qual a coluna r1 existe em R e S com o mesmo domínio.

FASE 6: Exploração

Operação de Intersecção

EXEMPLOS:

- Quais os nomes dos médicos que já deram consultas?

Nomes dos Médicos

```
SELECT f.nome FROM funcionarios f NATURAL JOIN medicos m;
```

Médicos que deram consultas

```
SELECT nr_mec_medico FROM consultas;
```

```
SELECT f.nome FROM funcionarios f NATURAL JOIN medicos m
WHERE m.nr_mec IN (SELECT nr_mec_medico FROM consultas);
```

```
SELECT f.nome FROM funcionarios f NATURAL JOIN medicos m
WHERE EXISTS (SELECT * FROM consultas c WHERE c.nr_mec_medico = m.nr_mec);
```

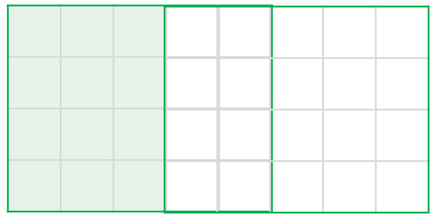
```
SELECT f.nome FROM funcionarios f NATURAL JOIN medicos m
WHERE m.nr_mec = ANY (SELECT nr_mec_medico FROM consultas);
```

```
SELECT f.nome FROM funcionarios f NATURAL JOIN medicos m
INNER JOIN consultas c ON c.nr_mec_medico = m.nr_mec;
```

FASE 6: Exploração

Operação de Diferença

A operação de Diferença, é uma operação entre duas relações compatíveis R e S que gera uma relação com esquema igual a R que contém todas as tuplas pertencentes a R, mas não pertencentes a S. Diz-se que duas relações são compatíveis se possuírem o mesmo grau (nº de colunas) e se as colunas correspondentes forem do mesmo domínio (tipo de dados).



diferença

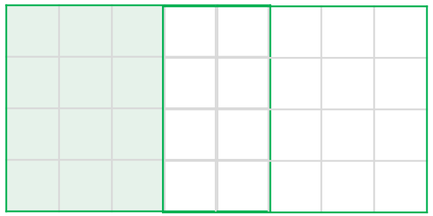
$$SELECT * FROM R$$
$$EXCEPT/MINUS$$
$$SELECT * FROM S;$$

O EXCEPT e o MINUS não são suportados pelo MySQL.

No entanto, esta operação pode ser representada usando outras operações como not exists, not in, all, e join.

FASE 6: Exploração

Operação de Diferença



diferença

```
SELECT * FROM R
EXCEPT|MINUS
SELECT * FROM S;
```

O EXCEPT e o MINUS não são suportados pelo MySQL.

```
SELECT * FROM R
WHERE r1 NOT IN
(SELECT r1 FROM S);
```

```
SELECT * FROM R
WHERE NOT EXISTS
(SELECT * FROM S
WHERE R.r1 = S.r1);
```

```
SELECT * FROM R
WHERE r1 <> ALL
(SELECT r1 FROM S );
```

```
SELECT * FROM R
LEFT JOIN S USING
(r1) WHERE S.id is
NULL;
```

NOTA: Na qual a coluna r1 existe em R e S com o mesmo domínio.

FASE 6: Exploração

Operação de Diferença

EXEMPLOS:

- Quais os nomes dos médicos que nunca deram consultas?

Nomes dos Médicos

$$SELECT\ f.nome\ FROM\ funcionarios\ f\ NATURAL\ JOIN\ medicos\ m;$$

Médicos que deram consultas

$$SELECT\ nr\_mec\_medico\ FROM\ consultas;$$

$$SELECT\ f.nome\ FROM\ funcionarios\ f\ NATURAL\ JOIN\ medicos\ m$$
$$WHERE\ m.nr\_mec\ \mathbf{NOT\ IN}\ (SELECT\ nr\_mec\_medico\ FROM\ consultas);$$

$$SELECT\ f.nome\ FROM\ funcionarios\ f\ NATURAL\ JOIN\ medicos\ m$$
$$WHERE\ \mathbf{NOT\ EXISTS}\ (SELECT\ *\ FROM\ consultas\ c\ WHERE\ c.nr\_mec\_medico\ =\ m.nr\_mec);$$

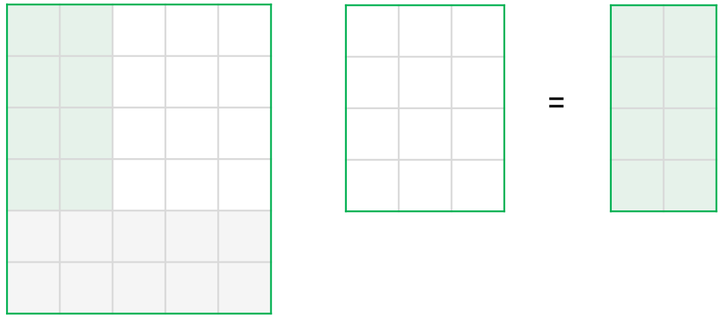
$$SELECT\ f.nome\ FROM\ funcionarios\ f\ NATURAL\ JOIN\ medicos\ m$$
$$WHERE\ m.nr\_mec\ <>\ \mathbf{ALL}\ (SELECT\ nr\_mec\_medico\ FROM\ consultas\ c);$$

$$SELECT\ f.nome\ FROM\ funcionarios\ f\ NATURAL\ JOIN\ medicos\ m$$
$$\mathbf{LEFT\ JOIN}\ consultas\ c\ ON\ c.nr\_mec\_medico = m.nr\_mec\ WHERE\ c.nr\_episodio\ IS\ NULL;$$

FASE 6: Exploração

Operação de Divisão

A operação de Divisão, é uma operação entre duas relações R e S, na qual as colunas de S devem constituir um subconjunto das colunas de R. Esta operação gera uma relação com esquema igual a todas as colunas de R que não são de S, através da seleção de tuplas da relação R que façam referência a todas as tuplas da relação S.



divisão/quociente

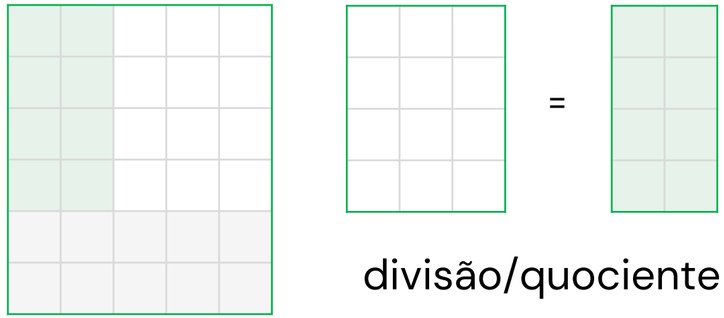
$$SELECT * FROM R$$
$$DIVIDE$$
$$SELECT * FROM S;$$

O DIVIDE não é suportado pelo MySQL.

No entanto, esta operação pode ser representada usando outras operações como not exists, not in, count, etc.

FASE 6: Exploração

Operação de Divisão



divisão/quociente

$$SELECT * FROM R$$
$$DIVIDE$$
$$SELECT * FROM S;$$

O DIVIDE não é suportado pelo MySQL.

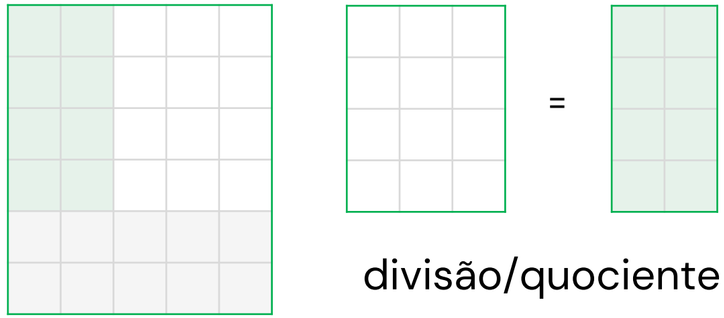
NOTA:
- a tabela RS relaciona as tabelas R e S;
- r1 existe em R e RS com o mesmo domínio;
- s1 existe em S e RS com o mesmo domínio

$$SELECT * FROM R WHERE \textbf{NOT EXISTS}$$
$$(SELECT * FROM S WHERE S.s1 \textbf{NOT IN}$$
$$(SELECT s1 FROM RS WHERE R.r1 = RS.r1));$$

$$SELECT * FROM R WHERE \textbf{NOT EXISTS}$$
$$(SELECT * FROM S WHERE \textbf{NOT EXISTS}$$
$$(SELECT * FROM RS WHERE R.r1 = RS.r1$$
$$AND S.S1 = RS.S1));$$

FASE 6: Exploração

Operação de Divisão



divisão/quociente

```
SELECT * FROM R
DIVIDE
SELECT * FROM S;
```

O DIVIDE não é suportado pelo MySQL.

NOTA:
- a tabela RS relaciona as tabelas R e S;
- r1 existe em R e RS com o mesmo domínio;
- s1 existe em S e RS com o mesmo domínio

```
SELECT * FROM R WHERE
(SELECT COUNT(*) FROM S WHERE NOT EXISTS
(SELECT * FROM RS WHERE R.r1 = RS.r1
AND S.S1 = RS.S1)) = 0;
```

```
SELECT R2 FROM R
JOIN RS USING (R1)
JOIN S USING (S1)
GROUP BY R2
HAVING COUNT (DISTINCT(RS.S1))
= SELECT COUNT(S1) FROM S));
```

FASE 6: Exploração

Operação de Divisão (÷/)

- Quais os nomes dos pacientes que já foram atendidos por todos os médicos de Obstetrícia?

1º – NOT EXISTS + NOT IN

```
SELECT p.nome FROM pacientes p WHERE NOT EXISTS (
        SELECT * FROM medicos m NATURAL JOIN especialidades e WHERE e.des_especialidade='Obstetrícia'
        AND m.nr_mec NOT IN (
                SELECT nr_mec_medico FROM consultas c WHERE p.nr_sequencial=c.nr_sequencial
        )
);
```

2º – NOT EXISTS + NOT EXISTS

```
SELECT p.nome FROM pacientes p WHERE NOT EXISTS (
        SELECT * FROM medicos m NATURAL JOIN especialidades e WHERE e.des_especialidade='Obstetrícia' AND
        NOT EXISTS (
                SELECT * FROM consultas c WHERE p.nr_sequencial=c.nr_sequencial AND m.nr_mec =
                c.nr_mec_medico
        )
);
```

FASE 6: Exploração

Operação de Divisão (÷/)

- Quais os nomes dos pacientes que já foram atendidos por todos os médicos de Obstetrícia?

3º – COUNT

```
SELECT p.nome FROM pacientes p
JOIN consultas c USING (nr_sequential)
JOIN medicos m ON c.nr_mec_medico = m.nr_mec
JOIN especialidades e USING (cod_especialidade)
WHERE e.des_especialidade='Obstetrícia'
GROUP BY p.nome HAVING COUNT(DISTINCT c.nr_mec_medico) = (SELECT COUNT(nr_mec) FROM medicos
NATURAL JOIN especialidades WHERE des_especialidade='Obstetrícia');
```

ou

```
SELECT p.nome FROM pacientes p WHERE (
    SELECT COUNT(*) FROM medicos m NATURAL JOIN especialidades e WHERE e.des_especialidade='Obstetrícia'
    AND NOT EXISTS (
        SELECT * FROM consultas c WHERE p.nr_sequencial=c.nr_sequencial AND m.nr_mec=c.nr_mec_medico
    )
) = 0;
```

FASE 6: Exploração

Vistas

Uma vista é uma tabela virtual derivada de uma ou mais tabelas ou vistas existentes. É definida por uma query SQL e tem a aparência de uma tabela, mas não armazena nenhum dado. Em vez disso, recupera os dados dinamicamente das tabelas/vistas subjacentes sempre que é consultada.

As vistas apresentam várias vantagens, entre as quais:

- **Simplificação de querys complexas** -> permite encapsular consultas SQL complexas usadas com frequência num único objeto reutilizável.

- **Segurança dos dados** -> permite conceder aos utilizadores acesso à vista enquanto restringe o acesso direto às tabelas subjacentes, garantindo a confidencialidade e a integridade dos dados.

- **Otimização de desempenho** -> permite pré-computar e armazenar em cache os resultados de consultas SQL complexa, melhorando o desempenho ao evitar a necessidade de recalcular o mesmo conjunto de resultados repetidamente.

FASE 6: Exploração

Vistas

Na aula de **Modelação Física**, na fase "**Representar os dados derivados**", vimos que no Hospital Portucalense, os seguintes atributos derivados poderiam ser calculados através de uma VIEW.

- **custo total (Consultas):** preço consulta + preço do procedimento + preço de equipamentos

- **custo final (Consultas):** **se** comparticipação = co-pagamento **então** custo final = custo total * (1-cobertura)
  caso contrário custo final = custo total

Vamos primeiro criar uma view que permita obter o **custo total** de uma consulta. Para isso precisamos de:

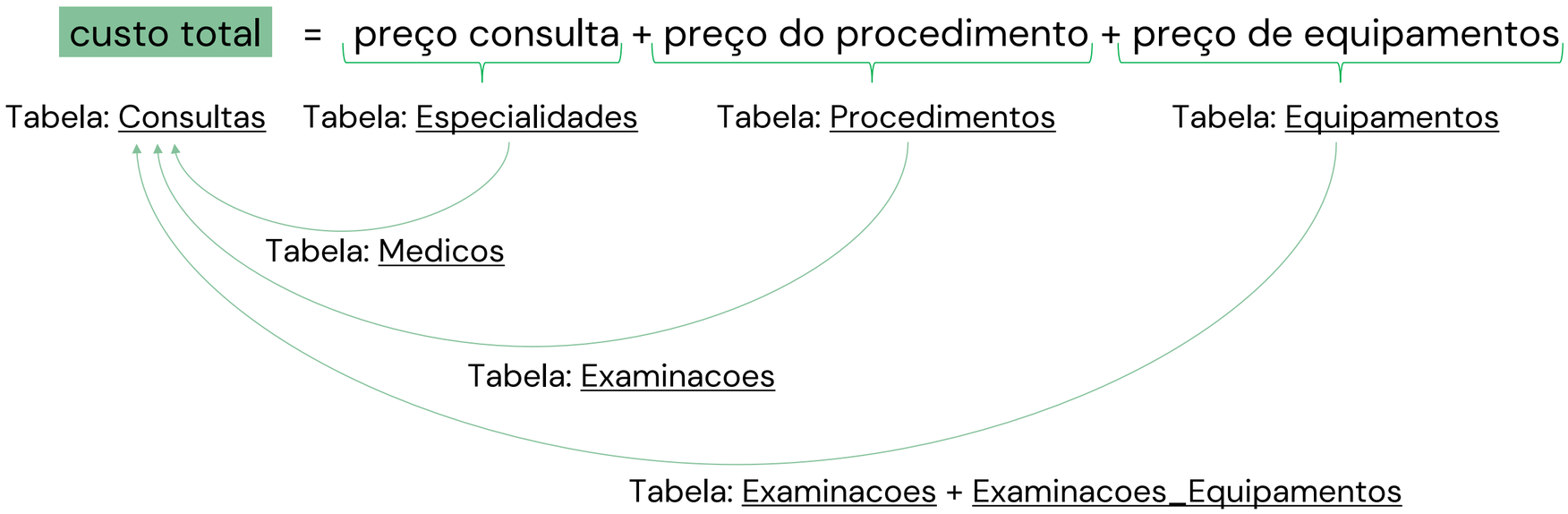
1) Criar a query SQL que permita fazer este cálculo;

2) Criar a view integrando a query SQL criada anteriormente.

FASE 6: Exploração

Vistas

1) Criar a query SQL que permita fazer este cálculo;



FASE 6: Exploração

Vistas

1) Criar a query SQL que permita fazer este cálculo;

custo total = preço consulta + preço do procedimento + preço de equipamentos

```
SELECT c.nr_episodio, c.dta_ini, c.dta_fim, c.nr_sequencial, m.nr_mec,
e.preco_consulta AS "Preço Consulta",
SUM(p.preco) AS "Preço Procedimento",
SUM(eq.preco) AS "Preço Equipamento",
(e.preco_consulta + SUM(p.preco) + SUM(eq.preco)) AS “Custo Total”
FROM consultas c
LEFT JOIN medicos m ON c.nr_mec_medico = m.nr_mec
LEFT JOIN especialidades e ON m.cod_especialidade = e.cod_especialidade
LEFT JOIN examinacoes ex ON c.nr_episodio = ex.nr_episodio
LEFT JOIN procedimentos p ON ex.cod_proc = p.cod_proc
LEFT JOIN examinacoes_equipamentos ee ON ex.id_examinacao = ee.id_examinacao
LEFT JOIN equipamentos eq ON ee.id_equipamento = eq.id_equipamento
GROUP BY c.nr_episodio;
```

✗ *O valor do preço total dos procedimento vai aparecer multiplicado pelo número de equipamentos utilizados.*

FASE 6: Exploração

Vistas

1) Criar a query SQL que permita fazer este cálculo;

custo total = preço consulta + preço do procedimento + preço de equipamentos

```
SELECT c.nr_episodio, c.dta_ini, c.dta_fim, c.nr_sequencial, m.nr_mec,
e.preco_consulta AS "Preço Consulta",
proc.custos AS "Preço Procedimento",
SUM(eq.preco) AS "Preço Equipamento",
(e.preco_consulta + proc.custos + SUM(eq.preco)) AS “Custo Total”
FROM consultas c
LEFT JOIN medicos m ON c.nr_mec_medico = m.nr_mec
LEFT JOIN especialidades e USING (cod_especialidade)
LEFT JOIN (
        SELECT ex.nr_episodio, SUM(p.preco) AS custos FROM examinacoes ex INNER JOIN procedimentos p
        ON ex.cod_proc = p.cod_proc GROUP BY ex.nr_episodio)
AS proc ON c.nr_episodio = proc.nr_episodio
LEFT JOIN examinacoes ex ON c.nr_episodio = ex.nr_episodio
LEFT JOIN examinacoes_equipamentos ee ON ex.id_examinacao = ee.id_examinacao
LEFT JOIN equipamentos eq ON ee.id_equipamento = eq.id_equipamento
GROUP BY c.nr_episodio;
```

Para garantir que o preço total dos procedimentos não é multiplicado pelo número de equipamentos, é necessário introduzir uma subquery que calcula a soma dos preços dos procedimentos antes de unir com os equipamentos.

FASE 6: Exploração

Vistas

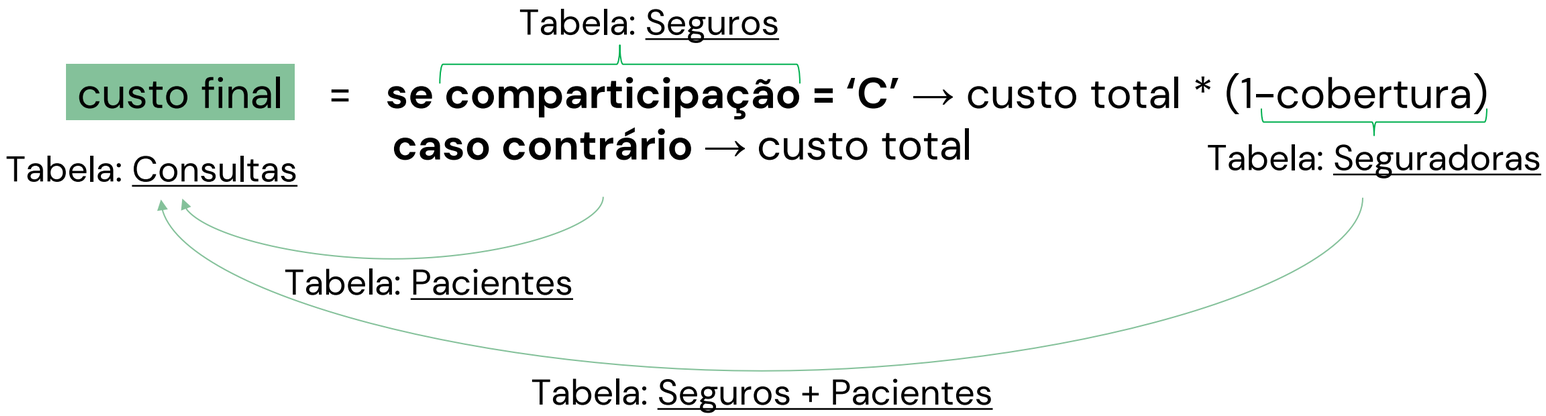
2) Criar a view integrando a query SQL criada anteriormente.

```
CREATE VIEW vwfaturacao AS
SELECT c.nr_episodio, c.dta_ini, c.dta_fim, c.nr_sequencial, m.nr_mec,
e.preco_consulta AS "Preço Consulta",
proc.custos AS "Preço Procedimento",
SUM(eq.preco) AS "Preço Equipamento",
(e.preco_consulta + proc.custos + SUM(eq.preco)) AS “Custo Total”
FROM consultas c
LEFT JOIN medicos m ON c.nr_mec_medico = m.nr_mec
LEFT JOIN especialidades e USING (cod_especialidade)
LEFT JOIN (
        SELECT ex.nr_episodio, SUM(p.preco) AS custos FROM examinacoes ex INNER JOIN procedimentos p
        ON ex.cod_proc = p.cod_proc GROUP BY ex.nr_episodio)
AS proc ON c.nr_episodio = proc.nr_episodio
LEFT JOIN examinacoes ex ON c.nr_episodio = ex.nr_episodio
LEFT JOIN examinacoes_equipamentos ee ON ex.id_examinacao = ee.id_examinacao
LEFT JOIN equipamentos eq ON ee.id_equipamento = eq.id_equipamento
GROUP BY c.nr_episodio;
```

FASE 6: Exploração

Vistas

1) Criar a query SQL que permita fazer este cálculo;



FASE 6: Exploração

Vistas

1) Criar a query SQL que permita fazer este cálculo;

custo final = **se comparticipação = 'C'** → custo total * (1-cobertura)
caso contrário → custo total

```
SELECT c.nr_episodio, c.dta_ini, c.dta_fim, c.nr_sequencial, c.nr_mec_medico, c.custo_total,
s.comparticipacao, ss.cobertura,
CASE
        WHEN s.comparticipacao='C' THEN c.custo_total * (1 - ss.cobertura)
        ELSE c.custo_total
END AS custo_final
FROM consultas c
INNER JOIN pacientes p USING (nr_sequencial)
LEFT JOIN seguros s USING (nr_apolice)
LEFT JOIN seguradoras ss USING (id_seguradora)
GROUP BY c.nr_episodio;
```

FASE 6: Exploração

Vistas

Juntando à query anterior

```
SELECT c.nr_episodio, c.dta_ini, c.dta_fim, c.nr_sequencial, m.nr_mec, s.comparticipacao, ss.cobertura,
e.preco_consulta AS "Preço Consulta", proc.custos AS "Preço Procedimento", SUM(eq.preco) AS "Preço Equipamento",
(e.preco_consulta + proc.custos + SUM(eq.preco)) AS "Custo Total",
CASE
        WHEN s.comparticipacao='C' THEN ROUND((e.preco_consulta + proc.custos + SUM(eq.preco)) * (1 -
ss.cobertura),2)
        ELSE ROUND((e.preco_consulta + proc.custos + SUM(eq.preco)),2)
END AS "Custo Final“
FROM consultas c
INNER JOIN pacientes pac USING (nr_sequencial)
LEFT JOIN seguros s USING (nr_apolice)
LEFT JOIN seguradoras ss USING (id_seguradora)
LEFT JOIN medicos m ON c.nr_mec_medico = m.nr_mec
LEFT JOIN especialidades e USING (cod_especialidade)
LEFT JOIN (   SELECT ex.nr_episodio, SUM(p.preco) AS custos FROM examinacoes ex INNER JOIN procedimentos p ON
ex.cod_proc = p.cod_proc GROUP BY ex.nr_episodio) AS proc ON c.nr_episodio = proc.nr_episodio
LEFT JOIN examinacoes ex ON c.nr_episodio = ex.nr_episodio
LEFT JOIN examinacoes_equipamentos ee ON ex.id_examinacao = ee.id_examinacao
LEFT JOIN equipamentos eq ON ee.id_equipamento = eq.id_equipamento
GROUP BY c.nr_episodio;
```

FASE 6: Exploração

Vistas

2) Criar a view integrando a query SQL criada anteriormente.

```
CREATE VIEW vwfaturacao AS
SELECT c.nr_episodio, c.dta_ini, c.dta_fim, c.nr_sequencial, m.nr_mec, s.comparticipacao, ss.cobertura, e.preco_consulta AS "Preço Consulta", proc.custos AS "Preço Procedimento", SUM(eq.preco) AS "Preço Equipamento", (e.preco_consulta + proc.custos + SUM(eq.preco)) AS "Custo Total",
CASE WHEN s.comparticipacao='C' THEN ROUND((e.preco_consulta + proc.custos + SUM(eq.preco)) * (1 - ss.cobertura),2)
ELSE ROUND((e.preco_consulta + proc.custos + SUM(eq.preco)),2)
END AS "Custo Final“
FROM consultas c INNER JOIN pacientes pac USING (nr_sequencial)
LEFT JOIN seguros s USING (nr_apolice)
LEFT JOIN seguradoras ss USING (id_seguradora)
LEFT JOIN medicos m ON c.nr_mec_medico = m.nr_mec
LEFT JOIN especialidades e USING (cod_especialidade)
LEFT JOIN ( SELECT ex.nr_episodio, SUM(p.preco) AS custos FROM examinacoes ex INNER JOIN procedimentos p ON ex.cod_proc = p.cod_proc GROUP BY ex.nr_episodio) AS proc ON c.nr_episodio = proc.nr_episodio
LEFT JOIN examinacoes ex ON c.nr_episodio = ex.nr_episodio
LEFT JOIN examinacoes_equipamentos ee ON ex.id_examinacao = ee.id_examinacao
LEFT JOIN equipamentos eq ON ee.id_equipamento = eq.id_equipamento
GROUP BY c.nr_episodio;
```

FASE 6: Exploração

Vistas

É possível realizar processos de consulta sobre as vistas criadas:

-- Liste as consultas e respetivos custos.
SELECT * FROM vwfaturacao;

-- Liste os pacientes que tiveram consultas no Hospital Portucalense, apresentando-os por ordem decrescente de lucro trouxeram maior lucro.
SELECT nr_sequencial as Paciente,
COALESCE(SUM(`Custo Total`), 0) AS "Custo Total",
COALESCE(SUM(`Custo Final`),0) AS "Custo Final“
FROM vwfaturacao
GROUP BY nr_sequencial
ORDER BY `Custo Total` DESC;

Para remover a vista:

DROP VIEW vwfaturacao;

SQL AVANÇADA

PREPARED STATEMENTS

Uma *query* pré-compilada permite a criação de declarações SQL para posterior execução, otimizando a execução de consultas repetitivas sem variação sintática, com dinamismo apenas nos parâmetros.

```
-- Preparar a query pré-compilada
PREPARE <prepared_stmt_name> FROM '<comando sql a executar de forma dinâmica*>';

-- Atribuir os valores aos parâmetros (caso existam)
SET @var1 = <value>;
SET @var2 = <value>;
...

-- Executar a query pré-compilada
EXECUTE <prepared_stmt_name> [USING @val1, @val2, ...];

-- Desalocar uma query pré-compilada
{DEALLOCATE | DROP} PREPARE <prepared_stmt_name>;
```

* O caractere "**?**" – *ponto de interrogação* – é utilizado em substituição de dados para indicar a espera de um parâmetro.

O maior benefício é a **velocidade na execução dos comandos SQL**, pois, após preparada, a declaração SQL é armazenada de forma pré-compilada no servidor da BD, sendo "*parseada*" apenas uma vez, mesmo que executada várias vezes.

SQL AVANÇADA

PREPARED STATEMENTS

EXEMPLO:

```
-- Preparar a query pré-compilada
PREPARE ps_medicos_por_especialidade FROM
'SELECT f.nome FROM funcionarios f
INNER JOIN medicos m USING(nr_mec)
INNER JOIN especialidades e USING(cod_especialidade)
WHERE e.des_especialidade = ?';

-- Atribuir os valores aos parâmetros (caso existam)
SET @especialidade = 'Neurologia';

-- Executar a query pré-compilada
EXECUTE ps_medicos_por_especialidade USING @especialidade;

-- Executar a query pré-compilada usando outros valores
SET @especialidade = 'Cardiologia'; EXECUTE ps_medicos_por_especialidade USING @especialidade;
```

SQL AVANÇADA

Variáveis, Constantes e Atribuições

Variáveis e **constantes** têm que ser declaradas, através da instrução DECLARE. Se uma variável for declarada sem especificar um valor padrão, o seu valor será NULL.

```
DECLARE nome tipo_de_dados(tamanho) [NOT NULL] [DEFAULT default_value];
```

```
DECLARE nome CONSTANT tipo_de_dados(tamanho);
```

As variáveis podem ser **atribuídas** de duas maneiras:

- usando a instrução de atribuição normal através da instrução SET

```
SET nome = valor/expressão;
```

- como resultado de uma instrução SQL SELECT

```
SELECT valor/expressão INTO nome FROM table_name WHERE condition;
```

SQL AVANÇADA

Estruturas básicas de programas

As rotinas pertencem à BD e são armazenadas no servidor. Os três componentes principais são:

Procedimentos (*Stored Procedures*)

- Blocos de código armazenados na BD que são pré-compilados.
- Podem operar nas tabelas da BD e retornar escalares ou conjuntos de resultados.

Funções (*Functions*)

- Pode ser usado como uma função interna para fornecer capacidade expandida às instruções SQL.
- Pode receber qualquer número de argumentos e retornar um único valor ou conjuntos de resultados.

Gatilhos (*Triggers*)

- É despoletado em resposta a operações de BDs padrão numa tabela específica.
- Pode ser usado para executar automaticamente operações de BDs adicionais quando ocorre o evento de acionamento.

SQL AVANÇADA

PROCEDURES

Um **procedimento** é uma coleção de instruções SQL pré-compiladas armazenadas na BD que mais tarde podem ser invocadas. Um procedimento que se invoca a ele mesmo é chamado de procedimento armazenado recursivo.

Do ponto de vista empresarial, é geralmente necessário executar tarefas específicas como limpeza da base de dados, processamento de folhas de pagamento, entre outras. Essas tarefas envolvem várias instruções SQL que podem ser agrupadas numa única tarefa, criando um procedimento armazenado na base de dados.

Os procedimentos podem ser invocados usando gatilhos (*triggers*), outros procedimentos e aplicações em Java, Python, PHP, etc.

Síntaxe para remover um procedimento:

```
DROP PROCEDURE [ IF EXISTS ]  <procedure_name>;
```

SQL AVANÇADA

PROCEDURES

Síntaxe para criação de um procedimento:

```
DELIMITER &&
CREATE PROCEDURE <procedure_name> ([IN | OUT | INOUT] parameter_name parameter_datatype)
BEGIN
    Declaration_section
    Executable_section
END &&
DELIMITER;
```

Nome do parâmetro

Tipo de parâmetro

Tipo de dados e tamanho do parâmetro

IN – É o modo padrão, permite passer parâmetros de entrada.

Tipo de Parâmetro

OUT – É usado para passar um parâmetro como saída. O seu valor pode ser alterado dentro do procedimento armazenado e o valor alterado (novo) é passado de volta para o programa que invoca o procedimento.

INOUT – É uma combinação dos modos IN e OUT.

SQL AVANÇADA

Procedimento sem parâmetros

Podemos criar um procedimento sem parâmetros. A rotina a baixo descrita é um procedimento que retorna todos os médicos do hospital.

```
DELIMITER &&
CREATE PROCEDURE GetMedicos()
BEGIN
    SELECT * FROM medicos;
END &&
DELIMITER;
```

- CALL GetMedicos();

Procedimentos (*Procedure*)

Procedimento com parâmetros IN

Sabemos que a tabela de preços está constantemente sujeita a alterações. De seguida, encontra-se uma rotina para atualizar o preço de um determinado procedimento clínico.

```
DELIMITER &&
CREATE PROCEDURE UpdateProcedimento (IN proc INT, IN new_preco DECIMAL(5,2))
BEGIN
    UPDATE procedimentos SET preco = new_preco where cod_proc=proc;
END &&
DELIMITER;
```

- CALL UpdateProcedimento(8, 10.15);

Procedimentos (*Procedure*)

Procedimento com parâmetros OUT

Sabemos que um hospital está constantemente sujeito a auditorias e a processos de avaliação dos serviços de saúde prestados. De seguida, encontra-se uma rotina para consultar o número total de consultas num determinado ano.

```
DELIMITER &&
CREATE PROCEDURE GetConsultasYear (IN ano INT, OUT total_consultas INT)
BEGIN
    SELECT count(*) INTO total_consultas FROM consultas WHERE YEAR(dta_ini)=ano;
END &&
DELIMITER;
```

```
CALL GetConsultasYear(2020, @total_consultas);
SELECT @total_consultas AS total_consultas_2020;
```

Procedimentos (*Procedure*)

Procedimento com parâmetros INOUT

No exemplo a seguir, encontra-se uma rotina que aceita um parâmetro IN e outro INOUT. Este procedimento, incrementa o contador de acordo com o valor específico no parâmetro inc.

```
DELIMITER &&
CREATE PROCEDURE Contador (IN inc INT, INOUT contador INT)
BEGIN
    SET contador = contador + inc;
END &&
DELIMITER;
```

```
SET @contador = 1;
CALL Contador(1, @contador); -- 2
SELECT @contador;
```

SQL AVANÇADA

FUNCTIONS

Uma **função** é um programa armazenado que devolve **um único valor**. Tipicamente, usam-se funções para encapsular fórmulas ou regras de negócio comuns que são **reutilizáveis** entre instruções SQL ou programas armazenados.

Síntaxe para criar uma função:

```
DELIMITER &&
CREATE FUNCTION <function_name> (parameter_name parameter_datatype)
RETURNS datatype [NOT] DETERMINISTIC
BEGIN
    Body_section
END &&
DELIMITER;
```

Nome do parâmetro

Tipo de dados e tamanho do parâmetro

Tipo de dados a retornar

Uma função **determinística** retorna sempre o mesmo resultado para os mesmos parâmetros de entrada.

Body_section → é preciso especificar pelo menos uma instrução RETURN.

Síntaxe para remover uma função:

```
DROP FUNCTION [ IF EXISTS ] <function_name>;
```

SQL AVANÇADA

FUNCTIONS

EXEMPLO → Função Idade

```
DELIMITER &&
CREATE FUNCTION idade (dta DATE)
RETURNS INT DETERMINISTIC
BEGIN
    RETURN TIMESTAMPPDIFF(YEAR, dta, CURDATE());
END &&
DELIMITER;
```

SQL AVANÇADA

TRIGGER

Um ***trigger*** é **invocado automaticamente** quando uma operação de alteração específica (instrução INSERT, UPDATE, ou DELETE) é executada sobre uma determinada tabela. Os *triggers* são úteis para tarefas como a aplicação de regras comerciais ou até para validação de dados quando inseridos na base de dados.

Síntaxe para criação de trigger:

```
DELIMITER &&
CREATE TRIGGER <trigger_name> {BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON <table_name> FOR EACH ROW
[{FOLLOWS | PRECEDES} existing_trigger_name]
BEGIN
    Body_section
END &&
DELIMITER;
```

Tempo de ação

Operação que ativa o trigger

Caso exista mais do que um trigger na mesma tabela com o mesmo evento e tempo de ação.

FOLLOWS permite que o novo trigger seja ativado após um trigger existente.
PRECEDES permite que o novo trigger seja ativado antes de um trigger existente.

Consegue aceder aos valores das colunas afetadas pela instrução DML. Os modificadores NEW e OLD permitem distinguir entre os valores antes e depois da operação de manipulação.

Síntaxe para remoção de trigger:

```
DROP TRIGGER [ IF EXISTS ] <trigger_name>;
```

SQL AVANÇADA

TRIGGER

EXEMPLO → Trigger para atualizar o preço_total e o preço_final da consulta antes da inserção de uma consulta

```
DELIMITER //
CREATE TRIGGER update_consultas_costs
BEFORE INSERT ON consultas
FOR EACH ROW
BEGIN
DECLARE v_custo_total DECIMAL(5,2);
DECLARE v_custo_final DECIMAL(5,2);
DECLARE v_cobertura DECIMAL(3,2);
DECLARE v_comparticipacao CHAR(1);
...
```

Declaração de variáveis

SQL AVANÇADA

TRIGGER

EXEMPLO → Trigger para atualizar o preço_total e o preço_final da consulta antes da inserção de uma consulta

```
...
SELECT s.comparticipacao, ss.cobertura INTO v_comparticipacao, v_cobertura
FROM pacientes p
LEFT JOIN seguros s ON p.nr_apolice=s.nr_apolice
LEFT JOIN seguradoras ss ON s.id_seguradora=ss.id_seguradora
WHERE p.nr_sequencial = NEW.nr_sequencial;
```

Preencher as variáveis comparticipacao e cobertura com os valores necessários

```
SELECT e.preco_consulta INTO v_custo_total
FROM medicos m
LEFT JOIN especialidades e ON m.cod_especialidade = e.cod_especialidade
WHERE m.nr_mec = NEW.nr_mec_medico;
...
```

Preencher a variável custo_total com o preço da especialidade da consulta que está a ser inserida

SQL AVANÇADA

TRIGGER

EXEMPLO → Trigger para atualizar o preço_total e o preço_final da consulta antes da inserção de uma consulta

```
...
IF v_comparticipacao = 'C' THEN
        SET v_custo_final = ROUND(v_custo_total * (1 - v_cobertura), 2);
ELSE
        SET v_custo_final = v_custo_total;
END IF;

SET NEW.custo_total = v_custo_total;
SET NEW.custo_final = v_custo_final;

END //
DELIMITER ;
```

Estabelecer a lógica de cálculo para o custo_final

Atribuir os valores ao custo_total e custo_final

NOTA: Como é um trigger BEFORE INSERT, não podemos fazer UPDATE consultas SET ... uma vez que a linha que queremos modificar ainda não existe. Então basta fazer SET NEW.col1 = val1

SQL AVANÇADA

HANDLER

Quando ocorre um erro é importante tratá-lo adequadamente, como continuar (CONTINUE) ou sair (EXIT) da execução do bloco de código atual e emitir uma mensagem de erro significativa. Para isso usa-se um *handler*, através da instrução DECLARE HANDLER.

Síntaxe:

Condição que ativa o handler

```
DECLARE action HANDLER FOR condition_value statement;
```

Tipo de acção — **Instrução a executar**

A instrução DECLARE CONDITION permite declarar uma condição de erro. Após a sua declaração, a declaração do *handler* pode se referir à *condition_name* em vez de se referir à *condition_value*.

```
DECLARE condition_name CONDITION FOR condition_value;
```

SQL AVANÇADA

HANDLER

Condição que ativa o handler

```
DECLARE action HANDLER FOR condition_value statement;
```

Tipo de acção — **Instrução a executar**

A **acção** pode ser de um dos seguintes tipos:

- **CONTINUE:** a execução do bloco de código envolvente continua.
- **EXIT:** a execução do bloco de código envolvente termina.

A **condição de ativação** pode ser:

- Um código de erro do MySQL;
- Um valor SQLSTATE padrão: SQLWARNING , NOTFOUND ou SQLEXCEPTION.
- Uma condição nomeada associada a um código de erro MySQL ou a um SQLSTATE através da instrução instrução DECLARE CONDITION.

A **instrução a executar** pode ser uma instrução simples ou uma instrução composta delimitada pelas palavras-chave BEGIN e END.

SQL AVANÇADA

HANDLER

```
DELIMITER $$
CREATE PROCEDURE InserirFarmaco ( IN codigo INT, IN farmaco VARCHAR(45), IN desc VARCHAR(150))
BEGIN
DECLARE EXIT HANDLER FOR 1062 SELECT 'Duplicate keys error encountered' Message;
DECLARE EXIT HANDLER FOR SQLEXCEPTION SELECT 'SQLException encountered' Message;
DECLARE EXIT HANDLER FOR SQLSTATE '23000' SELECT 'SQLSTATE 23000' ErrorCode;
INSERT INTO Farmacos(id_farmaco,nome,descricao) VALUES(codigo,farmaco,desc);
SELECT COUNT(*) FROM Farmacos WHERE id_farmaco = codigo;
END$$
DELIMITER ;
```

Sair/terminar em caso de duplicação de chaves

SQL AVANÇADA

TRANSACTIONS

A transação permite executar um conjunto de operações para garantir que a BD nunca contém o resultado de operações parciais. Num conjunto de operações, se uma delas falhar, ocorre a reversão para restaurar a base de dados ao seu estado original. Se nenhum erro ocorrer, todo o conjunto de instruções será confirmado na BD.

- Para iniciar uma transação, usa-se a instrução START TRANSACTION. O BEGIN ou BEGIN WORK são os *aliases* da instrução START TRANSACTION.
- Para **confirmar** a transação atual e tornar as suas alterações permanentes, usa-se a instrução **COMMIT**.
- Para **reverter** a transação atual e cancelar as suas alterações, usa-se a instrução **ROLLBACK**.
- Para desabilitar ou habilitar o modo de auto_commit usa-se a instrução SET auto_commit.

```
SET autocommit = 0; ou SET autocommit = OFF;
```

SQL AVANÇADA

➡ PROCEDURE WITH TRANSACTIONS AND HANDLERS

```
DELIMITER $$
CREATE PROCEDURE ProcTransacExemplo ( IN codigo INT, IN farmaco VARCHAR(45), IN desc VARCHAR(150), OUT res VARCHAR(100)
BEGIN
DECLARE erro INT DEFAULT 0;
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET erro=1;
START TRANSACTION;
INSERT INTO Farmacos(id_farmaco,nome,descricao) VALUES(codigo,farmaco,desc);
SELECT COUNT(*) FROM Farmacos WHERE id_farmaco = codigo;
IF erro = 1 THEN
    ROLLBACK;
    SET res = 'Transação abortada.';
    LEAVE InserirFarmaco;
END IF;
(...)
END$$
DELIMITER ;
```

SQL AVANÇADA

EVENTS

Um evento é uma tarefa que é executada num **horário específico**. Um evento pode conter uma ou mais instruções que são armazenadas na BD e executadas no cronograma especificado.

Os eventos podem ser criados para **execução única** ou para **determinados intervalos**. Para executar o evento de forma repetitiva, a cláusula EVERY pode ser usada.

Síntaxe para criação de um event

```
CREATE EVENT [IF NOT EXISTS] <event_name>
  ON SCHEDULE <time_stamp> | EVERY <interval> <quantity>
  DO <event_body>;
```

Síntaxe para remoção de um event

```
DROP EVENT [IF EXIST] <event_name>;
```

SQL AVANÇADA

EVENTS

EXEMPLO → relatório mensal com data, total de pacientes, total de consultas e total de examinações

```
CREATE EVENT monthly_report
ON SCHEDULE EVERY 1 MONTH
STARTS CURRENT_TIMESTAMP
DO BEGIN
    INSERT INTO Reports (report_date, total_patients, total_appointments, total_examinations)
    SELECT CURDATE(), COUNT(DISTINCT p.nr_sequencial), COUNT(DISTINCT c.nr_episodio), COUNT(e.id_examinacao)
    FROM consultas c
    LEFT JOIN pacientes p ON c.nr_sequencial = p.nr_sequencial
    LEFT JOIN examinacoes e ON c.nr_episodio = e.nr_episodio;
    WHERE YEAR(c.dta_ini) = YEAR(current_date()) AND MONTH(c.dta_ini) = MONTH(current_date())
END;
```

Próxima aula: Exploração da BD