



SISTEMAS DE BANCO DE DADOS 1

AULA 10

Linguagem SQL - Restrições **MySQL** e **ORACLE**

Vandor Roberto Vilardi Rissoli



APRESENTAÇÃO

- *Alter Table*
- *Primary, Foreign Key, Unique e Check*
- *Select* com algumas cláusulas
 - *Distinct e Alias*
 - *Group by*
 - *Order by*
- Referências



SQL

Instrução ALTER TABLE

Por meio desta instrução é possível :

- adicionar um novo atributo
- modificar uma atributo existente
- eliminar um atributo da relação

ALTER TABLE <tabela> [tarefa] (informações da tabela);

ALTER TABLE ESTADOS

ADD (cidade varchar(10)); -- inclui novo atributo na tabela

ALTER TABLE ESTADOS -- altera atributo

MODIFY (cidade varchar(30));

ALTER TABLE ESTADOS -- apaga atributo

DROP COLUMN cidade); 

Eliminação de um atributo por vez e a relação se mantém sempre com um atributo

SQL

Restrições de Integridade

As instruções de integridade são criadas junto com a tabela ou podem ser incluídas depois da sua criação, por meio da instrução **ALTER TABLE**.

Um tipo de restrição, que já foi estudado, é a criação de um atributo obrigatório (**NOT NULL**), em que os atributos devem possuir um valor, não podendo ser NULOS para serem inseridos na tabela.



SQL

Assim, observe então uma instrução que poderia alterar um atributo que aceitava valores nulos, sendo alterado para ser obrigatório na tabela CARRO.

ALTER TABLE CARRO

MODIFY COLUMN chassi VARCHAR(17) NOT NULL;

Para o sucesso dessa alteração todas as tuplas dessa tabela, que já pode ter dados armazenados exigiram que o atributo chassi possua dados, caso contrário o SGBD não conseguirá realizar esta modificação na tabela CARRO.



SQL

A identificação e a criação de uma chave primária consiste também em uma restrição. Esta restrição (chave primária) cria um recurso relevante na relação para se estabelecer o relacionamento seguro entre relações.



Cada relação só pode possuir uma chave primária, no qual esta chave é formada por um atributo ou conjunto de atributos da relação (chave simples ou composta).



SQL

Esta restrição impõe a **exclusividade** do atributo (ou conjunto de atributos), além de assegurar que **nenhum** atributo desta chave possua valor NULO.

CONSTRAINT <nome> **PRIMARY KEY** (<atributos>);

Cria-se a chave primária junto com a criação da tabela:

```
CREATE TABLE ESTADO (  
    sigla varchar(2) NOT NULL ,  
    nome varchar(20) ,  
    CONSTRAINT ESTADO_PK PRIMARY KEY(sigla)) ;
```

Tabela criada.

← resultado

Por meio desta instrução foi criada uma relação de nome ESTADO, com uma chave primária de nome ESTADO_PK, que sempre terá um valor único na relação.

SQL

A criação pode ser feita para uma relação que já exista, como:

```
DROP TABLE ESTADO;           -- apaga a relação já criada
CREATE TABLE ESTADO (
    sigla varchar(2) NOT NULL,
    nome varchar(20) );
```

Na instrução a seguir é criada a chave primária para a relação ESTADO, por meio de um **ALTER TABLE**.

```
ALTER TABLE ESTADO
    ADD CONSTRAINT ESTADO_PK PRIMARY KEY (sigla);
```

Com estas instruções será criada uma relação de nome ESTADO que possui a chave primária **sigla**, em que esta chave primária é responsável pela identificação única de uma tupla na relação ESTADO. Veja o exemplo acima:

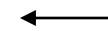
SQL

Após a criação da relação ESTADO, inicia-se o processo de construção do BD, em que os dados estarão sendo inseridos.

```
INSERT INTO ESTADO VALUES('AM', 'AMAZONAS');  
INSERT INTO ESTADO VALUES('SP','SAO PAULO');  
INSERT INTO ESTADO VALUES('SC','SANTA CATARINA');  
INSERT INTO ESTADO VALUES('AM', 'AMAPA');
```

*ERRO na linha 1:

Restrição exclusiva (SYS.ESTADO_PK) violada



resultado

```
INSERT INTO ESTADO VALUES('AM', 'AMAPA');
```

Como pode ser observado acima, a inserção do estado AMAPA não pode ser feita, pois por algum motivo tentou-se inserir, equivocadamente, este estado com a SIGLA de 'AM'. Este atributo é chave primária da relação ESTADO e já apresenta o estado AMAZONAS com esta SIGLA.

SQL

Realmente a SIGLA AM representa o estado do Amazonas, por isso pode-se imaginar que a inserção iria ser feita de forma errônea, por erro de digitação ou por erro no processo de coleta, ou ainda formação dos dados que compõe esta relação. Com o estabelecimento da restrição na relação, ela identificou o erro que se estaria cometendo e não permitiu o armazenamento, até então solicitado.

Outra restrição importante que utiliza as chamadas chaves candidatas, atributo que possui valor único, mas que não é chave primária, é identificado como atributo de valor único (**UNIQUE**).



SQL

A restrição **UNIQUE** requer que cada valor do atributo (ou conjunto de atributos) seja exclusivo, ou seja, não se repita (duplicado), como na restrição de chave primária (PRIMARY KEY).

Essa restrição também é chamada de chave exclusiva, quando envolve um único atributo, ou chave exclusiva composta quando envolve mais que um atributo.

Os valores nulos são aceitos nesta restrição, a não ser que o atributo envolvido na restrição (ou os atributos) esteja com a restrição **NOT NULL** descrita.



SQL

A restrição **UNIQUE** pode ser criada em nível de coluna ou tabela, porém uma chave exclusiva composta só pode ser feita no nível de tabela (sigla padrão **UK**).

Essa restrição pode ser criada juntamente com a relação ou depois da mesma, por exemplo:

```
DROP TABLE ESTADO;           -- apagando a relação já criada
```

```
CREATE TABLE ESTADO (  
    sigla varchar(2) NOT NULL,  
    nome varchar(20));
```

-- Inserindo a restrição em uma relação já existente

```
ALTER TABLE ESTADO  
    ADD CONSTRAINT ESTADO_UK UNIQUE (nome);
```

→ O **MySQL** e o **ORACLE** criam um **índice** exclusivo sobre a chave exclusiva (**unique**) para este tipo de restrição.

SQL

Outra restrição de integridade relacionada a criação e manipulação de chaves é a restrição de integridade referencial, que cria uma chave estrangeira em uma outra relação, ou na própria relação, fazendo o auto-relacionamento.



Por meio de uma CHAVE ESTRANGEIRA, que é formada por um ou mais atributos, é criado um relacionamento com a chave primária de uma outra relação, ou com a mesma relação (auto-relacionamento).

SQL

A relação que possui a chave primária é chamada de relação pai (ou *master*), enquanto a relação com a chave estrangeira é chamada de filho (ou *detail*). Quando existir um **auto-relacionamento** em uma relação, esta relação é pai e filho ao mesmo tempo.



Suponha que seja necessário conhecer as cidades da Federação e a qual Unidade da Federação (estado) cada uma destas cidades pertencem. Para isso será necessário armazenar dados de cada uma das cidades que formam o BD.

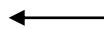


SQL

Elaborando o projeto do BD que será necessário para atender a necessidade do sistema desejado, constatou-se que os dados relacionados a relação CIDADE deverão ser formados pelos atributos de nome da cidade, quantidade de habitantes, estado que ela pertence, além de um código que a identifique unicamente dentro da relação (sua chave primária), formando assim a relação CIDADE.

```
CREATE TABLE CIDADE (  
    idCidade      int          NOT NULL,    -- número inteiro  
    cidade        varchar(40) NOT NULL,  
    qtdeHabitante bigint,  
    estado        varchar(2),  
    CONSTRAINT CIDADE_PK PRIMARY KEY (idCidade));
```

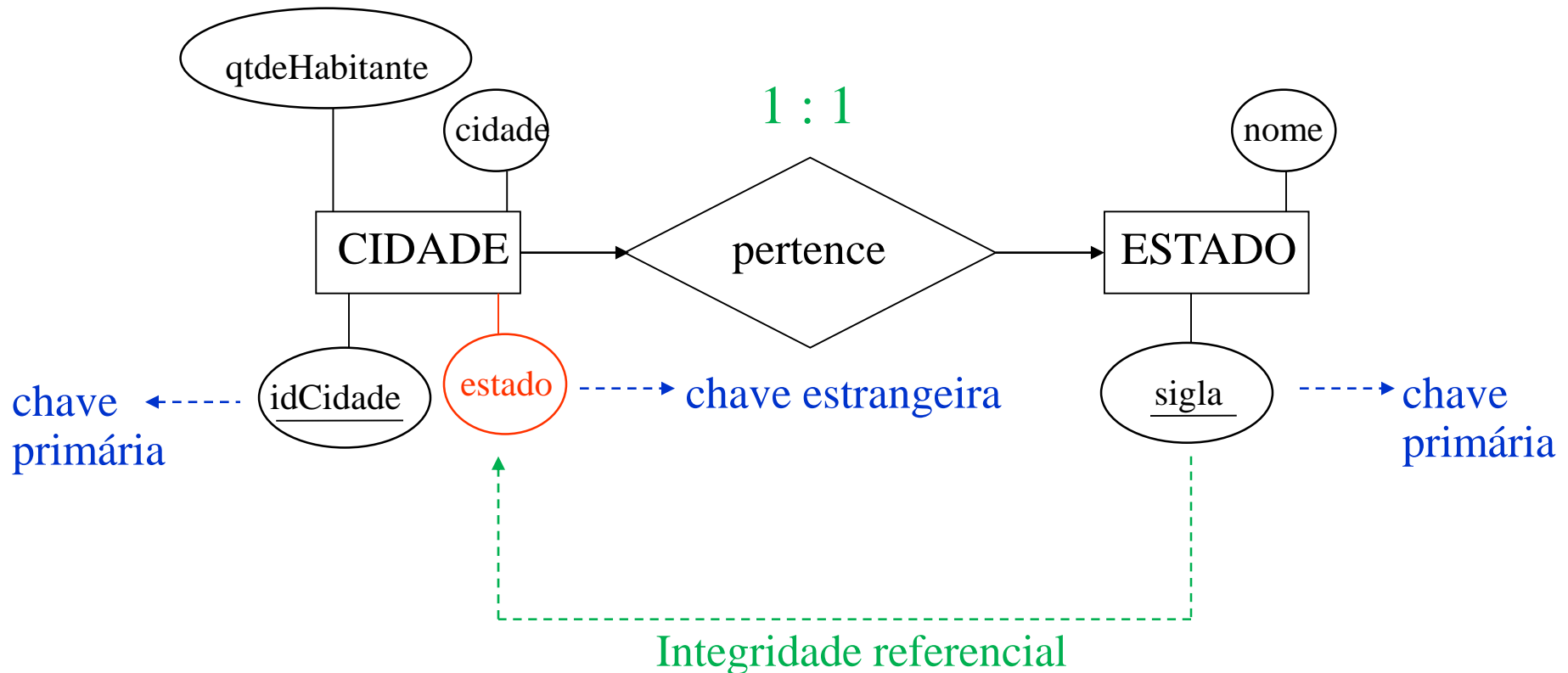
Tabela criada.



resultado

SQL

Elaborando o ME-R e o seu respectivo DE-R para esta situação tem-se:



O mapeamento deste modelo gerará as seguintes relações, contendo algumas restrições, tais como:

SQL

Analizando parte do referido ME-R anterior e mapeando sua tabela (ou relação) correspondente a implementação de CIDADE no Modelo Relacional se teria:

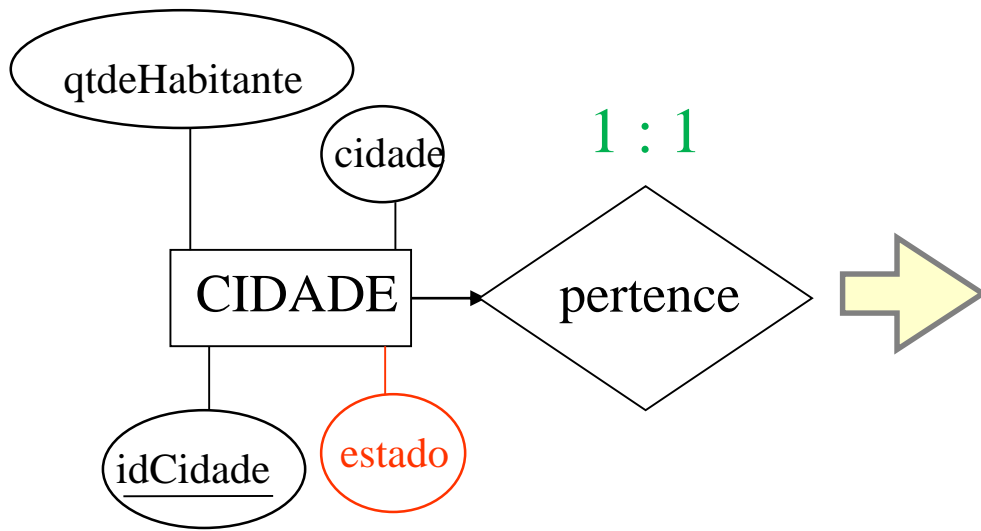


Tabela **CIDADE** (

idCidade int NOT NULL,
cidade varchar(40) NOT NULL,
qtdeHabitante bigint,
estado varchar(2),
sigla varchar(2) NOT NULL,

CONSTRAINT **CIDADE_PK** PRIMARY
KEY (idCidade));

Por que a tabela CIDADE tem **dois atributos** para armazenar o mesmo dados (sigla do estado que a cidade pertence)?



SQL

Corrigindo o ME-R que não tem atributo **estado**, já presente pelo relacionamento **pertence**, e como exemplo da chave primária, a chave estrangeira também poderia ser criada junto com a relação ou após sua criação com **ALTER**.

```
CREATE TABLE CIDADE (  
    idCidade      int NOT NULL,  
    cidade        varchar(40) NOT NULL,  
    qtdeHabitante bigint,  
    sigla         varchar(2),  
    CONSTRAINT CIDADE_PK PRIMARY KEY (idCidade),  
    CONSTRAINT CIDADE_ESTADO_FK FOREIGN KEY (sigla)  
        REFERENCES ESTADO (sigla) );
```

OU

```
ALTER TABLE CIDADE  
    ADD CONSTRAINT CIDADE_ESTADO_FK  
    FOREIGN KEY (sigla) REFERENCES ESTADO (sigla);
```

SQL

Com esta especificação de restrição uma cidade deverá possuir uma sigla de um estado já cadastrado na relação ESTADO (relação pai). Caso seja informada uma sigla não existente na relação pai a inserção não será realizada.

Ou a chave primária esta especificada corretamente, criando a relação entre as relações, ou ela deve ser nula na relação filho.

```
INSERT INTO CIDADE VALUES(1,'MARÍLIA',300000,'SP');  
INSERT INTO CIDADE VALUES(1,'MARÍLIA',300000,'RR');
```

*** ERRO na linha 1:**

**Restrição de integridade (SYS.ESTADO_CIDADE_FK)
violada - chave-pai não localizada**

← resultado



SQL

```
INSERT INTO CIDADE VALUES(1,'MARÍLIA',300000, 'SC');
```

***ERRO na linha 1:**



resultado

Restrição exclusiva (SYS.ID_CIDADE_PK) violada

```
INSERT INTO CIDADE VALUES(2,'MARÍLIA',300000, null);
```

```
INSERT INTO CIDADE VALUES(3,'MARÍLIA',300000, 'SC');
```

```
SELECT * FROM CIDADE; -- consultando todos dados inseridos
```

IDCIDADE NOME

QTDE_HABITANTE SIGLA

resultado

| | | | |
|---|---------|--------|----|
| 1 | MARÍLIA | 300000 | SP |
| 2 | MARÍLIA | 300000 | |
| 3 | MARÍLIA | 300000 | SC |

```
INSERT INTO ESTADO VALUES('PR', 'PARANA');
```

```
INSERT INTO CIDADE VALUES(4,'MARINGA',400000, 'PR');
```

```
DELETE FROM CIDADE WHERE IDCIDADE = 2;
```

```
DELETE FROM CIDADE WHERE SIGLA = 'SC';
```

SQL

A chave estrangeira é sempre definida na relação filho e a relação contendo o atributo referenciado (ou atributos) é a relação pai.

Para a definição desta restrição as palavras reservadas **FOREIGN KEY**, **REFERENCES** e **ON DELETE CASCADE** são empregadas, por exemplo:

```
CREATE TABLE CIDADE (  
    idCidade int NOT NULL,  
    cidade varchar(40) NOT NULL,  
    estado varchar(2),  
    CONSTRAINT CIDADE_PK PRIMARY KEY (idCidade),  
    CONSTRAINT CIDADE_ESTADO_FK FOREIGN KEY (estado)  
    REFERENCES ESTADO (sigla) ON DELETE CASCADE);
```

Identifica o(s) atributo(s) na relação filho

Identifica a relação pai e seu(s) atributo(s)

Identifica que as tuplas da relação filho também serão apagadas quando a tupla da relação pai for apagada

SQL

O comportamento de uma chave estrangeira (*FK*) deve sempre ser definido por seu projeto de implementação, **NÃO** ficando dependente de padrões que não sejam conhecidos pelo projetista ou de definições da Organização, a fim de que sua solução resolva o problema.

```
CREATE TABLE CIDADE (  
    idCidade int          NOT NULL,  
    cidade varchar(40)    NOT NULL,  
    estado varchar(2),  
    CONSTRAINT CIDADE_PK PRIMARY KEY (idCidade),  
    CONSTRAINT CIDADE_ESTADO_FK FOREIGN KEY (estado)  
        REFERENCES ESTADO (sigla) ON DELETE RESTRICT  
        UPDATE RESTRICT );
```

No exemplo acima as operações de **DELETE** e **UPDATE** deverão respeitar os aspectos de transações seguras e que mantêm a integridade da base de dados para acontecerem ou não.

SQL

Assim, o comportamento para **DELETE** e **UPDATE** serão sempre definidos em seu *script* de implementação do projeto, independente de qualquer outra definição que impeça a resolução necessária, podendo ainda **mesclar** comportamentos:

```
CREATE TABLE CIDADE (  
    idCidade int          NOT NULL,  
    cidade varchar(40)    NOT NULL,  
    estado varchar(2),  
    CONSTRAINT CIDADE_PK PRIMARY KEY (idCidade),  
    CONSTRAINT CIDADE_ESTADO_FK FOREIGN KEY (estado)  
        REFERENCES ESTADO (sigla) ON DELETE RESTRICT  
                                     UPDATE CASCADE );
```

DELETE RESTRICT = respeita transações seguras para integridade dos dados ou não realiza na *pai*;

UPDATE CASCADE = alteração na tabela *pai* transcorre também nas tuplas das tabelas *filhas*.

SQL

As opções de restrição relacionadas as tabelas relacionadas por uma chave estrangeira no **MySQL** variam conforme a necessidade da implementação, sendo possíveis:

- CASCADE: Atualiza ou exclui os registros da tabela filha automaticamente, ao atualizar ou excluir uma tupla da pai;
- RESTRICT: Rejeita a atualização ou exclusão de um registro da tabela pai, se houver registros na tabela filha;
- SET NULL: Define como NULL o valor do campo na tabela filha, ao atualizar ou excluir o registro da tabela pai;
- NO ACTION: Equivale à opção RESTRICT, mas a verificação de integridade referencial é executada após a tentativa de alterar a tabela. É a opção PADRÃO se nenhuma opção for definida na criação de chave estrangeira;
- SET DEFAULT: Define um valor padrão para coluna da tabela filha, aplicado quando um registro da tabela pai for atualizado ou excluído.

SQL

Uma restrição **CHECK** especifica uma condição que deverá ser satisfeita para cada tupla da relação, podendo ela ser definida a nível de atributo ou relação.

Verificar na versão 8
do MySQL

```
CREATE TABLE CIDADE (  
    idCidade      int          NOT NULL,  
    cidade        varchar(40) NOT NULL,  
    qtdeHabitante bigint,  
    estado        varchar(2),  
    CONSTRAINT CIDADE_PK PRIMARY KEY (idCidade),  
    CONSTRAINT HABITANTE_CK CHECK (qtdeHabitante > 0 ) );
```

Um único atributo pode possuir várias restrições **CHECK**, não havendo limite no número destas restrições que podem ser definidas em um atributo (**nível só de tupla**).

=> MySQL até a versão 5.7 ainda não tinha esta restrição implementada.

SQL

A restrição de **CHECK** em **ORACLE** pode usar as mesmas construções condicionais elaboradas para consultas (**SELECT**), existindo algumas exceções como:

- Não são permitidas referências às pseudocolunas CURRVAL, NEXTVAL, LEVEL e ROWNUM;

identificador inteiro do usuário corrente ↙

- Nem as chamadas as funções SYSDATE, UID,
 retorna o nome do usuário corrente no BD ← **USER** e **USERENV**; → retorna informações de uma sessão corrente

- Nem as consultas que se referem a outros valores em outras tuplas... mas para isso existem alternativas, entre elas em alguns SGBD trabalhar com **ASSERTION**



SQL

Uma outra expressão que pode ser usada na criação de relações é a **DEFAULT** que insere um valor no atributo quando este não for informado. Observe o exemplo:

```
CREATE TABLE CIDADE (  
    id                int NOT NULL,  
    cidade            varchar(40) NOT NULL,  
    aniversario       date DEFAULT CURDATE() NOT NULL ,  
    habitante         bigint,  
    CONSTRAINT CIDADE_PK PRIMARY KEY (id),  
    CONSTRAINT HABITANTE_CK CHECK (habitante > 0 ));
```

Não precisa ser
NOT NULL

↑

CURDATE → função de captura da data do sistema operacional (MySQL).

```
INSERT INTO CIDADE VALUES(1,'Brasília','2001-04-20', 2000000);
```

```
INSERT INTO CIDADE(id, cidade, habitante)
```

```
VALUES (2, 'São Paulo', 4000000);
```

-- Cadê a data obrigatória?

Consultando o Banco de Dados

Elabore uma consulta que permita visualizar todos os dados do esquema ESTADO e CIDADE.

Para que todos os dados das duas relações sejam mostrados é necessário realizar uma operação de junção entre elas.

Apesar de existir o relacionamento entre as duas relações, este relacionamento só será usado na apresentação dos dados (via SELECT por exemplo), se for especificada a expressão de junção na consulta, por exemplo:

```
SELECT nome, sigla FROM ESTADO;
```

```
SELECT idCidade, cidade, habitante, aniversario  
FROM CIDADE;
```

Estas consultas mostrarão todos os dados (tuplas) de cada relação. Porém a solicitação feita acima deseja apresentar todos estes dados já relacionados **em uma só consulta**.

SQL

Para isso se usa a **junção** entre as relações, que podem ser feitas para mais que 2 tabelas, caso exista necessidade.

```
SELECT nome, sigla, idCidade, cidade, qtdeHabitante, estado
FROM ESTADO, CIDADE -- consulta sobre duas relações
WHERE sigla = estado; -- realizando a junção entre as duas
```

A realização destas consultas (**SELECT**) deverá acontecer sobre a implementação das tabelas **ESTADO** e **CIDADE** que tem *FK* (trabalhar com **CIDADE** do *slide 22*).

Melhor organizando a apresentação para o usuário a ordenação por **sigla** primeiro e para as siglas iguais a ordenação alfabética por do nome da cidade.

```
SELECT nome, sigla, idCidade, cidade, qtdeHabitante, estado
FROM ESTADO, CIDADE -- realizando a junção entre as duas
WHERE sigla = estado -- ordenando de forma crescente a
ORDER BY sigla, cidade; -- apresentação do resultado
```

SQL

No intuito de evitar a compreensão equivocada do compilador SQL e a incorreta execução, ou mesmo o **erro** de compilação da declaração SQL desejada, será abordada algumas regras de sintaxe importantes para a escrita correta de uma instrução SQL.

→ Crie as 2 relações a seguir, respeitando as descrições:

RELAÇÃO ESTADO

sigla CHARACTER(2)

nome CHARACTER(20)

RELAÇÃO CIDADE

codigo NUMÉRICO(5)

nome CHARACTER(50)

sigla CHARACTER(2)

habitantes NUMÉRICO GRANDE

→ Agora, insira **5 tuplas** válidas na relação ESTADO e mais **10 tuplas** na relação CIDADE, no qual ao menos uma tupla seja de cada estado cadastrado na relação ESTADO.

SQL

- Realize as seleções (consultas) solicitadas sobre estes dados elaborando um único script com a documentação coerente e exigida em cada script **SQL**:
- A)** Projeção de sigla e nome do estado da sigla SP e DF;
 - B)** Selecione somente o nome e a sigla das cidades que são dos estados RJ, DF e GO;
 - C)** Selecione todas as cidades do **primeiro** estado que você cadastrou, mostrando somente o nome da cidade, o nome do estado e sua sigla;
 - D)** Selecione somente o nome e a sigla do estado que você cadastrou por **último** e todas as cidades cadastradas nele, mostrando o nome da cidade e a quantidade de habitantes em ordem crescente de nome de estado e também de nome de cidade para cada estado.


SQL

PADRÃO PARA ENTREGA DE CONSULTAS

SOLUÇÃO (demonstração sobre o item **A** que deve ser entregue em outro *script* específico no formato ou padrão exigido em Linguagem **SQL**).

A)

```
SELECT sigla, nome
FROM ESTADO
WHERE sigla = 'SP'
OR sigla = 'DF';
```

A vertical dashed line is positioned between the SQL keywords and the values/expressions. A red arrow points to the right above the line, and a blue arrow points upwards along the line, indicating the indentation structure of the query.

Lembrar de manter a organização (indentação) ou usar a tecla de atalho no **MySQL Workbench** **Ctrl + B**



SQL

O resultado padrão a ser apresentado por uma instrução **SELECT** exibe todos os atributos solicitados em todas as tuplas selecionadas, o que pode incluir **tuplas duplicadas**.

Caso isso não seja desejável por uma determinada consulta, o qualificador **DISTINCT** deve ser inserido na instrução **SELECT** que será executada. Este qualificador vem logo após a palavra reservada **SELECT**, no qual em seguida podem ser especificados todos os atributos que se deseja apresentar como resultado desta consulta.

Por exemplo: suponha que a relação cidade possua 6 tuplas, sendo 3 cidades do estado de São Paulo.

```
SELECT sigla  
FROM JOSE.CIDADE  
WHERE sigla = 'SP';
```

Esta consulta selecionará somente as cidades com a sigla do estado de São Paulo (SP), mostrando assim três tuplas **idênticas**.

SQL

Este tipo de consulta também poderia incluir o qualificador **ALL** no início do **SELECT**, porém esta qualificação já é padrão quando um qualificador não é especificado, pois ela estabelece que todas as tuplas selecionadas devem ser apresentadas.

```
SELECT ALL sigla  
FROM JOSE.CIDADE  
WHERE sigla = 'SP';
```

Para que os valores selecionados não sejam apresentados em duplicidade o qualificador **DISTINCT** deve ser colocado no local do **ALL**.

```
SELECT DISTINCT sigla  
FROM JOSE.CIDADE  
WHERE sigla = 'SP';
```

Esta consulta apresentará uma única tupla com o valor da sigla SP, apesar de existirem mais que uma tupla com esse valor.

SQL

Quando os atributos solicitados não formarem uma tupla **totalmente** única eles serão apresentados normalmente.

```
SELECT DISTINCT sigla, codigo  
FROM JOSE.CIDADE  
WHERE sigla = 'SP';
```

O resultado apresentado consistirá de todas as tuplas que possuem a **sigla** SP, com códigos diferentes, pois se existir alguma sigla igual com código igual o qualificador **DISTINCT** não permitirá a sua apresentação porque ele separa as projeções distintas e as apresenta, por meio da análise do resultado que será projetado (mostrado) pelo **SELECT** (**projeções iguais não são mostradas**).



SQL

CRIANDO GRUPOS DE DADOS

Tudo que foi feito até aqui tratava todos os dados de uma relação como um único grupo ou coleção de dados, mas às vezes é necessário dividir estas relações em grupos menores.

Procurando realizar estas divisões ou agrupamento nos dados se pode incluir a cláusula **GROUP BY** em uma instrução **SELECT**.

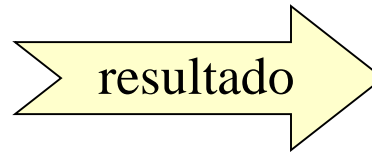
É possível aplicar uma cláusula **GROUP BY** em uma seleção e usar uma função de grupo para **sumarizar** (agrupar) os valores de cada um dos grupos formados (existentes na recuperação do **SELECT**).



SQL

Suponha que a consulta abaixo seja executada e o resultado seja o apresentado a seguir.

```
SELECT SIGLA, COUNT(sigla)
FROM CIDADE
GROUP BY sigla;
```



| SIGLA COUNT(SIGLA) | |
|--------------------|-------|
| ---- | ----- |
| CE | 3 |
| DF | 2 |
| SP | 5 |



O que isso significa?

Ou o que isso quer dizer?

A relação CIDADE foi dividida em três grupos por meio da sigla (estado), em que foram consideradas somente as siglas que estão cadastradas e foram recuperadas através da consulta (SELECT) realizada do atributo **sigla**, enquanto que a função **COUNT** mostra a quantidade de tuplas que existem em cada sigla (agrupamento).



SQL

Sempre que existir um ou mais atributos individuais na seleção, juntamente com a aplicação de funções de grupo, será necessário incluir estes atributos no grupo por meio da cláusula **GROUP BY**.

```
SELECT sigla, COUNT(sigla)
FROM CIDADE
GROUP BY sigla;
```

→ um atributo simples ou comum

A seleção sem especificação de nenhum atributo individual também é possível, mas será realizada por toda a relação se não for especificada a cláusula **GROUP BY**. Observe os resultados em seu SGBD aplicando as duas sugestões abaixo:

```
SELECT COUNT(sigla)
FROM CIDADE
GROUP BY sigla;
```

≠

```
SELECT COUNT(sigla)
FROM CIDADE;
```



SQL

O atributo especificado no **GROUP BY** não precisa estar no **SELECT**, mas normalmente seu resultado fica sem sentido devido a dificuldade de compreensão dos dados agrupados.

```
SELECT COUNT(sigla)
FROM CIDADE
GROUP BY sigla;
```

Pode também existir a necessidade de agrupar mais que um atributo, em que uma relação é agrupada primeiro por um atributo e dentro deste também é realizado o agrupamento por um outro atributo. Imagine o exemplo:

```
SELECT departamento, empregado, SUM(salario)
FROM EMPREGADO
GROUP BY departamento, empregado;
```

SQL

O uso da cláusula **WHERE** não pode ser aplicada na restrição de grupos, mas a inclusão da cláusula **HAVING** pode avaliar cada grupo e permitir a sua apresentação ou NÃO.

```
SELECT departamento, AVG(salario)
FROM EMPREGADO
GROUP BY departamento
HAVING AVG(salario > 2000);
```

Avaliação condicional
sobre o grupo

cláusula de avaliação onde os valores
que o satisfação são apresentados

→ Como é usada a cláusula **WHERE** na restrição de tuplas sobre atributos individuais, a cláusula **HAVING** também pode ser empregada, mas sobre funções de grupo da consulta.

SQL

Pode ser usada a cláusula **GROUP BY** sem uma função de grupo no **SELECT**. Porém, a apresentação dos resultados sempre acontecerão em ordem crescente quando um cláusula de grupo estiver na instrução.

Um **ORDER BY** pode ser inserido, mas por padrão a apresentação sempre será em **ordem crescente**. A aplicação do **ORDER BY** seria coerente para a alteração desta apresentação, no qual, por exemplo, o resultado desejado seria melhor empregado em uma ordem decrescente, ou mesmo sobre outros atributos da consulta.

```
SELECT departamento, AVG(salario)
FROM EMPREGADO
GROUP BY departamento
HAVING AVG(salario > 2000)
ORDER BY departamento;
```

Exercícios de Fixação

- 1) Elabore um conjunto de relações (tabelas) que permita armazenar e consultar os dados de **alunos** matriculados em um **curso**. Para isso, identifique quais são os dados necessários sobre o aluno e o curso, elaborando o ME-R e o respectivo DE-R do modelo proposto. Em seguida, realize o mapeamento do DE-R para as relações que formarão o BD com as restrições estudadas até o momento. Implemente-os no ambiente SQL e insira alguns dados (**mais que 4 tuplas por tabela**). Por fim, elabore as três consultas abaixo com um SELECT para cada uma:
 - a) todos os alunos matriculados;
 - b) todos os cursos disponíveis na instituição;
 - c) todos os alunos com a identificação completa do curso que ele esta fazendo na instituição.

Exercícios de Fixação

2.) Usando o ME-R elabore um projeto que implemente um BD para o controle de funcionários de uma empresa. Este controle visa gerenciar as atividades de cada funcionário que trabalha em um departamento específico dentro de uma única empresa. O motivo deste controle é reconhecer o envolvimento de cada departamento nos diferentes projetos que a empresa realiza, identificando, com isso, os departamentos envolvidos em cada projeto e os funcionários destes departamentos que fazem parte do projeto pesquisado. Os dados disponíveis dos projetos são sempre o nome, data prevista de início e do fim do projeto, além do funcionário responsável pelo projeto. Os departamentos possuem nome, além da identificação dos funcionários que o compõe (trabalham no departamento).

Exercícios de Fixação

... continuação do exercício 2

Os funcionários são contratados após a coleta dos dados pessoais (nome, CPF, carteira profissional, telefones e endereço) de cada um, além da identificação do cargo que ele estará ocupando, seu salário mensal e o departamento ao qual ele será alocado. Elabore também as seguintes **consultas** para atender a necessidade desta empresa:

- a) Identificação nominal do funcionário e seu endereço para uso do serviço de mala direta (etiqueta para correio);
- b) Os funcionários que pertencem a um departamento;
- c) Relatório de custos separados por departamento identificando os salários dos funcionários que formam tal departamento;
- d) Os departamentos envolvidos em um projeto específico;
- e) Os funcionários envolvidos em um projeto específico.

SQL – Conhecendo SEQUENCE

Criando Sequências

Alguns SGBDs usando de recursos diferentes para geração de valores sequências que podem ser empregados com segurança, por exemplo, como chaves primárias. Uma sequência (*sequence*) é um objeto do SGBD que gera números sequenciais e respeita as características fornecidas por parâmetros no momento de sua criação.

CREATE SEQUENCE <nome_da_sequência>

Parâmetros (**ORACLE**)

increment by / **start with** / **cycle** ou **nocycle**
maxvalue ou **nomaxvalue** / **minvalue** ou **nominvalue**
cache ou **nocache**

SQL – Conhecendo SEQUENCE

| OPÇÃO | Descrição |
|--------------------------|---|
| nome_da_sequência | Nome da sequencia, não podendo ser o mesmo de uma tabela |
| Increment by <i>n</i> | Especifica de quanto será o incremento ou decremento. O padrão é 1 |
| Start with <i>n</i> | Especifica o primeiro número a ser gerado. O padrão é 1. |
| Maxvalue <i>n</i> | Especifica o valor máximo que a sequência gerada pode atingir. O padrão é nomaxvalue, indo até 1027 |
| Minvalue <i>n</i> | Especifica o valor mínimo para as sequências que estiverem sendo decrementadas. É mutuamente exclusiva ao maxvalue. |
| Cycle nocycle | Indica que ao atingir o valor máximo a numeração continuará a partir do valor inicial. O default é nocycle. |
| Cache <i>n</i> nocache | Especifica quantos valores o bando de dados pré-aloca e mantém em memória. O padrão é 20. |



SQL – Conhecendo SEQUENCE

Exemplo:

```
CREATE SEQUENCE SEQ_PESSOA  
    minvalue 1  
    maxvalue 9999999999  
    start with 1  
    increment by 1  
    nocache  
    cycle ;
```

IMPORTANTE

O SGBD **MySQL** não trabalha com este tipo de objeto de BD, mas vários outros SGBD usam **SEQUENCE**

O exemplo acima só ilustra uma possível criação de SEQUENCE com muitos parâmetros em **ORACLE**, mas geralmente a criação é muito mais simples, pois recebe a maioria dos parâmetros com valor padrão do SGBD.

SQL – Conhecendo SEQUENCE

Exemplo de uso da SEQUENCE criada anteriormente no SGBD **ORACLE**:

```
SELECT SEQ_PESSOA.nextval
```

Caso for testar no banco de dados ou outro programa acrescente um **from dual** para obter o resultado.

```
SELECT SEQ.nextval FROM DUAL;
```

A recuperação do valor atual acontece com a substituição do **nextval** por **currval**

```
SELECT SEQ.currval FROM DUAL;
```



SQL – Conhecendo SEQUENCE

Usando como uma chave (**PK**) para inserir uma nova tupla na tabela **PESSOA**. Observe que um novo valor é solicitado para **SEQUENCE (SEQ_PESSOA)**:

```
INSERT INTO PESSOA
```

```
(idPessoa, nome, idade) VALUES (  
    SEQ_PESSOA.Nextval, 'José Silva', 25);
```

Para listar as *sequences* disponíveis para o usuário conectato pode ser usada a instrução **ORACLE** abaixo:

```
SELECT *  
FROM user_sequences;
```



SQL - Processando Transações

Além dos comandos DDL e DML vistos até aqui, uma outra categoria de comandos chamada de TPL (*Transaction Proccess Language* – Linguagem de Processamento de Transação) compõe a SQL.

Os comandos desta categoria são responsáveis pela gravação definitiva no banco de dados das instruções DDL e DML, além da recuperação do banco de dados, caso algo aconteça de errado.

Várias das instruções DDL são de salvamento automático (*autocommit*), portanto, uma vez executadas elas já estão implantadas no banco de dados. Por exemplo a instrução CREATE TABLE.



SQL

Salvando as Realizações

Após realizar algumas instruções sobre o banco de dados é necessário solicitar que ele armazene efetivamente (grave) o resultado de suas solicitações feitas por meio das instruções executadas (transações).

A efetiva realização solicitada pelas instruções **INSERT**, **DELETE** e **UPDATE** precisam ser salvas no banco de dados. Para que as suas ações (ou realizações) sejam salvas no banco de dados, é necessária a execução do comando **COMMIT**.

Por meio do comando **COMMIT** essas instruções DML são efetivadas (salvas) no banco de dados.



SQL

Salvando as suas realizações no BD será liberado todos os recursos exigidos até concluir a sua transação. Existe uma redução nos "**custos**" do acesso de outros usuários e a diminuição do seu tempo de espera.

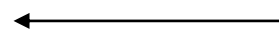
Outra característica importante para estar salvando sempre o que você estiver realizando, principalmente as DML, **diminui os riscos de perder** algo que terminou de fazer com tanto cuidado no BD.

Exemplo:

Após uma instrução DML correta executar o salvamento deve ser realizado nos SGBDs que não tem as DML com **autocommit**.

COMMIT;

Efetivação completada.



resultado

SQL

Durante o uso do SGBD pode acontecer a execução de uma instrução equivocada, ou problemas como a falta de energia no meio de uma transação, entre outros.

A restauração da situação do BD, antes de um salvamento ser executado, é possível pelo comando **ROLLBACK**. Ele retrocede o BD para situação que estava após o último salvamento (**COMMIT**) ser executado.

Exemplo:

ROLLBACK;

Restabelecimento completado.

← resultado

As instruções de **AUTOCOMMIT** não serão retornadas, pois assim que são executadas são salvas, **PERSISTINDO** no BD.

Vários SGBD permitem configurar a ação de **AUTOCOMMIT**.

SQL - Processando Transações

Usando o SGBD **MySQL** é interessante conhecer mais sobre a operação das instruções **COMMIT** e **ROLLBACK** na prática e com mais detalhes pela importância de cada uma delas.

Assim, assista ao vídeo indicado a seguir e realize as operações indicadas para maior compreensão e aprendizagem prática relacionada com estas instruções.

Endereço Virtual de Vídeo para Estudo prático

<https://youtu.be/FgeroBoBOAE?si=ekqlACF3w6PNcATV>



Referência de Criação e Apoio ao Estudo

Material para Consulta e Apoio ao Conteúdo

- ELMASRI, R. e Navathe, S. B., Fundamentals of Database Systems, Addison-Wesley, 3rd edition, 2000
 - Capítulo 8
- SILBERSCHATZ, A. & Korth, H. F., Sistemas de Banco de Dados.
 - Capítulo 4
- SUNDERRAMAN, R., Oracle Programming: A Primer, Addison Wesley, 1999.
 - Capítulo 2
- Universidade de Brasília (UnB Gama)
 - <https://sae.unb.br/cae/conteudo/unbfga/>
(escolha a disciplina **Sistemas de Banco de Dados 1**)
- Oracle: SQL Assertions/Declarative multi-row constraints
 - <https://community.oracle.com/ideas/13028>