contro f

Segunda - SQL - BOT

Terça- Python -

Quarta- Java - BOT

QUINTA - SQL -

Sexta - python - BOT

Sábado- java

**COMANDOS E SINTAXE PARA MANIPULAÇÃO DE BANCO DE DADOS**

**\***: selecionar todos atributos da tabela

**-CASE:** é a manifestação do if no SQL, exemplo de estrutura :

SELECT cor, (CASE

WHEN cor ="Vermelha" then "igual"

else "diferente"

END) as 'resultado' from cesta\_de\_frutas

obs: as aspas em resultado é opcional também demonstra sem colocar aspas, caso em mais de um when, se atente para a ordem das condições pois elas afetam na compilação/ também pode estar contido em um count, SUM , AVG, MAX MIN

COUNT: irá contar os registros com aquela condição vale pra caractere,texto e número

SUM e AVG: vale só para número, ira fazer uma soma dos registros que se encontram naquela condição ou tirar a média deles, depois do THEN virá 1, e else 0 indicando que 1 é verdadeiro e afeta na operação e 0 é falso pois sua entrada não afetará na operação. THEN tb pode retornar atributos

NOTA: EM CASO DE VÁRIAS CONDIÇÕES O SQL IRÁ DAR PRIORIDADE A PRIMEIRA CONDIÇÃO DE CIMA PARA BAIXO E ASSIM SUCESSIVAMENTE

**-Is:** é

**-Is not:** não é

**-not in**

**-In:** exemplo para melhor entendimento:

SELECT \* FROM sakila.payment Where amount IN (0.99, 2.99);

----é a mesma coisa que :

SELECT \* FROM sakila.payment Where amount= 0.99 OR amount=2.99;

**-Between :** entre

exemplo : between 0 and 5

outro exemplo:

SELECT \* FROM sakila.payment Where amount between 0 and 5;

///// é a mesma coisa de :

Select \*from sakila.payment Where amount > 0 and amount <5;

nota: ele considera os pontos de partida, ou seja, no caso acima 0 e 5 estão inclusos

**-Order by:** Ordem

**///asc:** ordem crescente

**///desc:** ordem decrescente

order by 1 ( primeiro atributo da consulta, depois do select)

**-Like :** Se parece com ou igual a. comando mais usado para variaveis caracters e texto. é segue de “%. assim irá trazer valores com o caracter entre as aspas e até os valores que contém os caracters “%commentaries%” retornar algo diferente de “commentaries” e de “%commentaries” e de “commentaries%”;

ele é case sensitive

**-ilike:** mesma coisa que o like mas não é case sensitive

**-CHARINDEX**: retorna a primeira posição de uma sequência de caracteres em uma determinada expressão

**-Where**: Somente onde

**-From**: qual tabela

**-Select:** consulte selecione

**-AS:** Mudança de nome do atributo

**-Distinct:** consulta os registros distintos, linhas diferentes entre se

**-alter table:** altera a tabela

**-add**: adiciona atributo/coluna

**-drop column**: tira atributo/coluna

**-add constraint**: adiciona relação regra

**-drop table**: remove tabela

**-drop database:** remove banco de dados

**-SUM:** soma

**-AVG:** Média

**-MAX:** Máximo

**-MIN:** Mínimo

**-STDDEV:** Calcula o desvio padrão

**-GROUP\_CONCAT:** CONCATENA valores de texto em grupos

**-Count(\*) ou (Atributo):** conta a quantidade de entidades/registros/linhas de cada tabela

ex: Select first\_name, count(\*) from actor

where first\_name="Nick"

order by 1;

**-group by:** agrupa registros repetidos, ou seja é como varios nomes em um so

usado para omitir repetições

group by 1( 1 coluna do select )

select nome,count(\*) as ‘qtd’ from autor group by nome

**-having:** função de agregação, estabelece uma condição, “ onde “ “a partir de”

exemplo: having count(\*) > 10, muito usado para especificar um ou mais grupos de repetições, não pode ser usado junto com o where.. vem depois de group by e antes de order by

**-INNER JOIN:** Junta tabelas em uma consulta select, estabelecendo condições entre elas, junta entidades , exemplo cliente e produto

**-CROSS JOIN:** AQUI nós teremos a comparação de cada linha de uma tabela com as das outra tabelas, o tabela final sempre terá como quantidade de linhas a multiplicação de linhas da tabela 1 com a 2 , e vice versa. nesse caso não é necessário usar ON. é um produto cartesiano

**-JOIN:** Junta tabelas em uma consulta select, sem precisar estabelecer condição, junta entidades , exemplo cliente e produto sem precisar usar o on ( geralmente não é bom usar, apresenta os dados da maneira errada) agora se você usar o ON com chave primary e estrangeira, ele vai executar a condição, e vai relacionar os dados na tabela de maneira coerente

**-RIGHT JOIN ou RIGHT OUTER JOIN:**  PRIORIZA A TABELA DA DIREITA E A INTERSECÇÃO, pode adicionar valores nulos

**-LEFT JOIN ou LEFT OUTER JOIN:**  PRIORIZA A TABELA DA ESQUERDA E A INTERSECÇÃO, pode adicionar valores nulos

**-IS NULL**: deleta intersecção da tabela (se colocado no where depois de qualquer join)

**-SOMENTE RIGHT JOIN :**

EX: select \* from t1

RIGHT JOIN t2

on t1.C=t2.C

where t1.C(key) IS NULL

**-SOMENTE LEFT JOIN:**

EX: select \* from t1

LEFT JOIN t2

on t1.C=t2.C

where t2.C(key) IS NULL

**-FULL OUTER JOIN:** PRIORIZA TODAS AS TABELAS E A INTERSECÇÃO também adiciona valores nulos

**-FULL OUTER JOIN SEM INTERSEÇÃO:**

EX:select \* from t1

FULL OUTER JOIN t2

on t1.C=t2.C

WHERE t1.C(key) IS NULL

OR

WHERE t2.C(key) IS NULL

**- FORMA EXTENSA DO FULL OUTER JOIN: ( com duplicadas,ou seja com interseção)**

select \* from t1

LEFT JOIN t2

on t1.C=t2.C

UNION ALL

select \* from t1

RIGHT JOIN t2

on t1.C=t2.C

**-SINTAXE COMPLETA DO FULL OUTER JOIN: (sem duplicadas,ou seja, sem interseção)**

select \* from t1

LEFT JOIN t2

on t1.C=t2.C

UNION

select \* from t1

RIGHT JOIN t2

on t1.C=t2.C

WHERE t1.C IS NULL;

**-ON:** ligação

**-AS:** Muda a forma que o atributo vai ser mostrado

**-TOP:** top1, top 2. serve para mostrar 1 , 2 registros de tal tabela e la vai

**-UNION:** une tabelas somente com colunas iguais e mesma quantidade de colunas, colunas iguais podem virar 1 coluna , ele remove linhas duplicadas, ou seja linhas com resultados iguais entre se considerando todas as colunas .

EX: select from UNION select from

**-UNION ALL**: mesma coisa do union mas ele não remove linhas duplicadas ( comandos union não são relacionamentos, eles apenas servem para juntar as tabelas sem envolver relação)

**-NATURAL JOIN:** É como o INNER JOIN, a diferença é que não deixa explícito o nome das colunas com a condição de relacionamento, ou seja não usa on e nem mostra as chaves estrangeiras e primárias. ele vai fazer a ligação procurando uma coluna de uma tabela com o mesmo nome de outra coluna em outra ou seja, quase colunas iguais nas duas tabelas. não é recomendado usar toda hora pois ele não vai ligar os dados de forma correta caso os registros sejam atualizados, principalmente se o banco tiver colunas de data de atualização update\_dt, que são as colunas que registram a mudança daquele registro(ou seja um registro que mostra quando aquele registro foi mudado para outro registro). O natural join requer um conhecimento aprofundado sobre o banco e as tabelas que você vai estar trabalhando, só é uma forma de facilitar sua consulta nesses casos. Caso esteja em um banco desconhecido e ou complexo é recomendável usar o inner join, que vai ligar chaves estrangeiras com primária de forma correta nas suas consultas.

/////////// exemplo: os registros da coluna update\_dt na tabela payment se referem a registros de valor diferente aos da tabela rental. ou seja o update\_dt das duas tabelas são diferentes e se referem a coisas diferentes, mesmo tendo ou não linhas iguais vão se referir a coisas diferentes e mesmo com linhas iguais tem que verificar se eles se referem a mesma coisa. /////////resumidamente o seu natural join para ser bem aplicado deve ser usado com tabelas que tenham somente 1 coluna em comum e que essa coluna/atributo seja a chave estrangeira

**-INNER JOIN () USING:** O USING serve para especificar qual coluna você que usar para linkar as tabelas, depois de escrever a coluna depois do using, ele irá procurar por essa colunas nessas duas tabelas, vendo se ela é comum as duas.

/////////resumidamente o seu using para ser bem aplicado deve constar a coluna que corresponde como chave estrangeira a outra tabela

**-EQUIVALÊNCIA DA CLÁUSULA ON:**

**exemplo:**

SELECT f.title

FROM film AS f

INNER JOIN film\_actor AS fa

ON fa.film\_id=f.film\_id

INNER JOIN actor AS a

ON a.actor\_id=fa.actor\_id

WHERE a.first\_name= "CHRISTIAN"

AND a.last\_name= "GABLE";

#### É EQUIVALENTE a

SELECT f.title

FROM film AS f

INNER JOIN film\_actor AS fa

ON fa.film\_id=f.film\_id

INNER JOIN actor AS a

ON a.actor\_id=fa.actor\_id

AND a.first\_name= "CHRISTIAN"

AND a.last\_name= "GABLE";

**SUBCONSULTA: EXEMPLO CONCEITUAL**

SELECT p.payment\_id, r\_c.customer\_id FROM payment p

LEFT JOIN ( SELECT r.rental\_id, c.customer\_id

FROM rental r

LEFT JOIN customer c on c.customer\_id=r.customer\_id) r\_c

on r\_c.rental\_id=p.rental\_id;

Perceba, aqui temos um select dentro desse select, a realidade é que aí é uma forma extensa, veja o select dentro dos parenteses como tabela, pois é isso que acontece dentro dos parênteses **,** a consulta dentro dele , gera uma nova tabela temporária, a qual não existe no banco de dados ou seja a tabela chama r\_c vai se relacionar com payment no final, o LEFT maior vai fazer com que os dados do payment sejam priorizados junto com a interseção com a tabela r\_c

**SUBCONSULTA:** também pode se usar IN e NOT IN. aqui a subconsulta só deve ter 1 coluna

exemplo : select in ( select

**WITH:** ele cria uma tabela que pode ser usado como subconsulta em outra consulta select

exemplo:

WITH filtro as (

SELECT customer\_id

FROM payment group by customer\_id

HAVING sum(amount)>200)

SELECT \* FROM customer c

WHERE customer\_id IN(

SELECT customer\_id FROM filtro);

//NOTE aqui nós nomeamos um select como se fosse uma tabela, no caso filtro representa nosso select condicional que estará logo dentro de outro select. isso serve para deixar os códigos mais organizados e menos extensos

**EXISTS OU NOT EXISTS**

SELECT \* FROM CUSTOMER c

WHERE EXISTS

( SELECT 1 FROM RENTAL r

WHERE r.customer\_id= c.customer\_id

and return\_date is null);

/// note o exist melhor a processamento do computador para compilar o código, aqui o exist funciona como a sequência binária de 1 e 0 para verificar se a condição é falsa ou verdadeira, o seu subselect pode conter qualquer número o que importa que é o que vem depois do FROM, o seu where vai funcionar como ferramenta para relacionar as tabelas com chave primária e estrangeira , cada tabela deve ser apelidada e caso tenha coloque a condição especificada depois

// a sub consulta relacionada aqui sempre depende da consulta principal pra ser executada então não é permitido usar o WITH

**CREATE TEMPORARY TABLE “nome” AS(codigo):**

São tabelas temporarias, visam ser usadas para facilitar demais consultas, elas são automaticamente deletadas quando você encerra seu aplicativo SQL, então não armazene nada importante

CREATE TEMPORARY TABLE clientes\_sem\_devolução AS

SELECT c.customer\_id, c.first\_name, c.last\_name, c.email

FROM customer c

WHERE c.customer\_id in

( Select customer\_id FROM rental

where return\_date is null);

SELECT \* from clientes\_sem\_devolução

**CREATE VIEW:**

a consulta em visualizações é igual a consulta em tabela, é como salvar um select e não criar uma tabela temporaria já que os dados não são salvos no banco

porém as visualizações são registradas no banco, elas podem ser vistas no DER, é perfeito para uso de atualizações

CREATE VIEW rv as

SELECT c.customer\_id, c.first\_name, c.last\_name, c.email, a.address

FROM customer c

LEFT JOIN address a on a.address\_id=c.address\_id;

SELECT \* from rv;

//// a create view, pode receber dados novos, ou seja as visualizações não são estáticas, enquanto tabela temporárias são. então é mais recomendável usar o create view para operações constantes e a longo prazo. enquanto tabelas temporárias são para análises situacionais mais específicas e momentâneas já que elas são deletadas ao fechar o app, enquanto as visualizações ficam permanentemente

**Consulta complexa:**

/\*

Consultas que visam analisar quantos % dos filmes estão faltando

\*/

WITH films as

(

SELECT film\_id, count(\*) as 'total\_inventory' from inventory

group by 1 order by 2 DESC

),

missing\_return as

(

SELECT film\_id, count(\*) as 'missing\_titles' from inventory

WHERE inventory\_id IN

( SELECT inventory\_id from rental

WHERE return\_date is null)

group by 1

)

SELECT title,

total\_inventory,

missing\_titles,

(missing\_titles/total\_inventory)\*100 as "perc\_faltam%"

from films

INNER JOIN missing\_return

USING (film\_id)

LEFT JOIN film

USING (film\_id)

ORDER BY total\_inventory DESC

**FUNÇÕES DE JANELA:** realiza operações ou cálculos de linha a linha , as linhas não são agrupadas em poucos valores. é como se ordenasse e agrupasse, a tabela sempre retornará o mesmo número de linhas da tabela original. não agrupa resultados

**-OVER (PARTITION BY):** separa por partes, ou em janelas aquele atributo

**-ROW\_NUMBER**: Ordena os resultados de forma crescente

-SELECT row\_number() OVER (ORDER BY) AS FROM # ordernação

-SELECT row\_number() OVER ( PARTITION BY) AS FROM # partição

-SELECT row\_number OVER ( ORDER BY PARTITION BY) AS FROM # partição e ordenação

**-LAG**: cria uma coluna que compara a linha atual com linhas anteriores

**-LEAD**: cria uma coluna que compara a atual com as proximas linhas

-SELECT LAG() OVER AS,

LEAD() OVER AS,

FROM

**-ROWS UNBOUNDED PRECEDING**: traga uma janela com essa partição de todas as linhas que estão antes da atual

ex:SELECT

tipo\_da\_fruta, peso,

SUM(PESO) OVER(

ROWS UNBOUNDED PRECEDING)

AS peso\_acumulativo

FROM cesta\_de\_frutas;

#ele irá somar todas as linhas antes da atual junto com ela, e trazer a soma acumulativa

ex:SELECT

tipo\_da\_fruta, peso,

SUM(PESO) OVER(

partition by tipo\_da\_fruta

ROWS UNBOUNDED PRECEDING)

AS peso\_acumulativo

FROM cesta\_de\_frutas;

#ele irá somar o peso acumulativo de cada tipo de fruta

**NTILE:** representa , declara quantos grupos irão ser criados para os nosso dados e os divide igualmente, e na ordernação é definido qual é a regra de cada grupo

ex:SELECT tipo\_da\_fruta,

peso,

NTILE(2) OVER (ORDER BY peso)

AS grupo\_fruta,

(CASE WHEN peso<=0.40 then "leve"

ELSE "pesada" END) AS tipo

FROM cesta\_de\_frutas

perceba que depois de usar NTILE , você pode adicionar o case WHEN com mais facilidade

**RANK:** lista os resultados repetidos a partir da sua posição,

Ou seja , empata os valores usando sua posição de chegada.

**DENSE\_RANK:** lista os resultados repetidos em sequência, empate os valores em ordem

exemplo para melhor entendimento :

SELECT

ROW\_NUMBER () OVER (ORDER BY tipo\_da\_fruta) AS funcao\_1,

RANK () OVER (ORDER BY tipo\_da\_fruta) AS funcao\_2,

DENSE\_RANK () OVER (ORDER BY tipo\_da\_fruta) as funcao\_3,

tipo\_da\_fruta

FROM cesta\_de\_frutas

ORDER BY tipo\_da\_fruta;

nota: função avg, max, min, e sum devem nas funções de janela devem ter o over e podem vir com partition by e também devem ser as primeiras a vir depois do select

**DATE()**: remove horários de colunas com variáveis do tipo data na consulta

ex: date(payment\_date)

**UNIX\_TIMESTAMP():** Permite que as colunas contem os segundos a partir de certa data, funciona apenas com colunas do tipo data

ex: UNIX\_timeSTAMP(payment\_date)

**NOW():** retorna a data e hora atual

**MONTH():** selecionar só os meses daquela coluna

**DAY:** selecionar só os dias daquela coluna

**YEAR :** Seleciona os anos daquela coluna

**UNIX\_TIMESTAMP():** retorna o timestamp atual

**CURDATE():** retorna a data atual

**CURRENT\_TIME():** retorna a hora atual

**DATEDIFF():** diferença entre duas data em dias

ex:SELECT DATEDIFF(CURDATE(), "1990-01-03") AS idade\_dias,

DATEDIFF(CURDATE(), "1990-01-03")/365 AS idade\_anos

**TIMEDIFF():** diferença entre dois períodos em horários

**DATE\_ADD():** adiciona um período a data

ex:SELECT CURDATE() AS hoje,

DATE\_SUB(CURDATE(), INTERVAL 1 DAY) AS ontem,

DATE\_ADD(CURDATE(), INTERVAL 1 DAY) AS amanha,

DATE\_SUB(CURDATE(), INTERVAL 1 MONTH) AS mes\_passado,

DATE\_ADD(CURDATE(), INTERVAL 1 WEEK) AS semana\_que\_vem,

DATE\_ADD(CURDATE(), INTERVAL 1 YEAR) AS ano\_que\_vem;

**DATE\_SUB():** subtrai um período da data

**EXTRACT:**  retorna um sub-valor da data

ex:SELECT EXTRACT(DAY FROM CURDATE() ) AS dia\_de\_hoje,

EXTRACT(YEAR FROM CURDATE()) AS ano\_atual,

YEAR(CURDATE()) AS ano\_atual;

**DATE\_FORMAT:**  arredonda a data e retorna outra data

ex:SELECT DATE\_FORMAT(CURDATE(), "%y-%m-01") AS mes\_atual,

DATE\_FORMAT(CURDATE(), "%y-01-01") AS ano\_atual;

ex2: SELECT EXTRACT(DAY FROM CURDATE() ) AS dia\_de\_hoje,

EXTRACT(YEAR FROM CURDATE()) AS ano\_atual\_extracao\_1,

YEAR(CURDATE()) AS ano\_atual\_extraca\_2,

MONTH(CURDATE()) AS mes\_atual\_extracao,

DAY(CURDATE()) AS dia\_atual\_extracao,

DATE\_FORMAT(CURDATE(), "%Y") AS ano\_atual\_extracao\_3,

DATE\_FORMAT(CURDATE(), "%y-%m-01") AS mes\_atual,

DATE\_FORMAT(CURDATE(), "%y-01-01") AS ano\_atual\_trunc;

**DATE\_TRUNC:** é mais simples usado em outros softwares de SQL

**INTERVAL:** determina o quantidade de intervalo entre as datas

**DAYOFWEEK / DAYOFMONTH /DAYOFYEAR:** determina o dia da semana, mês ou ano escolhido

ex:SELECT EXTRACT(WEEK FROM CURDATE()) AS semana\_atual,

DATE\_SUB(CURDATE(), INTERVAL DAYOFWEEK(CURDATE()) -1 DAY);

**WEEKDAY:** determina quando começou a semana, de 0 a 6 a partir da segunda feira

ex:SELECT EXTRACT(WEEK FROM CURDATE()) AS semana\_atual,

DATE\_SUB(CURDATE(), INTERVAL DAYOFWEEK(CURDATE()) -1 DAY),

DATE\_SUB(CURDATE(), INTERVAL WEEKDAY(CURDATE()) -1 DAY);

**UPPER:** Transforma todas as letras tipo texto da coluna em maiúsculas

**LOWER:** Transforma todas as letras do tipo texto da coluna em minúsculas

**INITCAP:** As primeiras letras de cada linha da coluna ficam maiúsculas

**CONCAT :** combina dois textos em uma só coluna

ex:SELECT CONCAT(first\_name, " ", last\_name) as full\_name

FROM staff;

**CONCAT\_WS:** combina vários textos em uma só coluna

ex:SELECT CONCAT\_WS(" ", first\_name, last\_name) as full\_name

FROM staff;

**REVERSE:** retorna um texto de trás pra frente

SELECT REVERSE(first\_name) AS nome\_contrario,

REVERSE(last\_name) AS sobrenome\_contrario

FROM sakila.staff;

**LEFT:** retorna caracteres à esquerda do texto a partir da posição especificada

all ex :

SELECT last\_name,

LEFT(last\_name, 3) AS esq,

RIGHT(first\_name, 2) AS dir,

MID(last\_name, 2, 6) AS meio

FROM staff;

**RIGHT:** retorna caracteres à direita do texto a partir da posição especificada

**MID ou SUBSTR:** retorna os caracteres na posição do meio, entre as posições especificadas

ex :SELECT title,

LEFT(title, 3),

substr(title, 2, 6),

RIGHT(LEFT(title, 7), 6)

FROM film;

ou

WITH aux as (SELECT title,

LEFT(title, 3),

substr(title, 2, 6),

LEFT(title, 7) as bom

FROM film)

SELECT RIGHT(BOM,6) from aux;

**REPLACE:** substitui um texto por outro texto novo na coluna e ou até characters

ex:SELECT last\_name,

REPLACE(last\_name, "e", "I") AS subst

FROM staff;

**TRIM:** remove espaço em branco antes e depois do texto

SELECT last\_name,

TRIM(last\_name) AS 2\_lados,

RTRIM(last\_name) AS direita,

LTRIM(last\_name) AS esqueda

FROM staff;

**RTRIM**: remove espaço em branco depois do texto

**LTRIM:** remove espaço em branco antes do texto

**LPAD:** acrescenta texto à esquerda da coluna

ex: SELECT first\_name,

LPAD(first\_name, "->") AS esqueda,

RPAD(first\_name, ">-") AS direita

FROM staff;

**RPAD:** acrescenta texto à direita da coluna

**EXPRESSÕES REGULARES CONCEITO:**  procurar linhas nas colunas determinadas conforme o padrão complexo de texto especificado , é usado no SQL e várias outras línguas e plataformas, até para criação de bots. primeiro argumento é a coluna, o segundo é a expressão. mais comumente utilizadas no where e having

**REGEXP / RLIKE / REGEXP\_LIKE() :** buscam por texto que correspondem à expressão regular (retornam TRUE or FALSE ) ou 1 e 0

ex: SELECT \* FROM film\_list

WHERE REGEXP\_LIKE(description, "Teacher");

ou

SELECT \* FROM film\_list

WHERE description LIKE "%Teacher%";

**REGEXP\_REPLACE()**: substitui texto que corresponda à expressão regular

**REGEXP\_SUBSTR():** retorna a parte do texto que corresponde à expressão regular

**COMANDOS E SINTAXE PARA CRIAÇÃO DE BANCO DE DADOS**

**CREATE TABLE**

**CREATE DATABASE**

**FOREIGN KEY**

**NOT NULL**

**NULL**

**PRIMARY KEY**

**AUTO INCREMENT**

**USE “”**

**UNIQUE:** EXCLUSIVIDADE

**///NÙMEROS**

**DECIMAIS(decimal/float)**

**BINÁRIOS(BOOL, Verdadeiro-1 , FALSO-0)**

**/////TEXTO**

**CHAR( número de caracters LIMITADO)**

**VARCHAR( número de caracteres ILIMITADO)**

**///DATAS**

**-DATA(DATE)**

**-DATE E HORA( DATETIME)**

**-TIMESTAMP ( Número de segundos depois de determinada data)**

**///TIPOS COMPLEXOS**

**-ESPACIAL(GIS)**

**-DICIONÁRIOS(JSON)**

**-VETORES(ARRAY)**

**-Identity:** serve pra fazer incrementações em colunas para variáveis inteiras

ex

codigo:

1

2

3

**COMMENT**: comentário da tabela em questão

**ALTER TABLE** nome\_da\_tabela (ação)**add column**

**RENAME COLUMN** a **TO** b

**MODIFY** a **DATETIME**

**TRUNCATE TABLE** nome da tabela: cuidado o comando deleta todos os dados da tabela e não é reversível

**DROP TABLE** nome da tabela : deleta a tabela selecionada

**DROP DATABASE** nome do banco: deleta o banco de dados

**INDICES:** Cria índices como se fosse um sumario de um livro, ajuda no processamento dos dados e na consulta dos dados é importante estabelecer índices ao criar o BD

**INSERT INTO VALUES:** adiciona dados em uma tabela por vez, também pode adicionar apenas nas colunas selecionadas , ao vez de adicionar dados em 4 colunas da tabela, apenas é adicionado em 2 colunas

ex: INSERT INTO tabela VALUES( valor1, valor 2, valor 3, valor 4)

ou INSERT INTO tabela (col1, col 2) VALUES( valor 1, valor 2)

ou INSERT INTO tabela VALUES( valor1, valor 2)

**INSERÇÃO POR CONSULTAS SELECTS:**

**INSERT INTO VALUES SELECT FROM** :mapeia o resultado de uma consulta para as colunas de uma tabela , os tipos de dados e número de colunas inserido precisa ser válido e correspondente

ex:INSERT INTOnome\_da\_tabela\_destino (col1,col2,col3) VALUES SELECTcol1, col2 , col3

FROM nome\_da\_tabela\_fonte;

//////////

**CREATE TEMPORARY AS TABLE SELECT FROM** : outra forma que também seleciona parte de uma tabela em outra só que temporária

CREATE TEMPORARY TABLEnome\_da\_tabela\_destino AS

SELECTcol1, col2, col3

FROMnome\_da\_tabela\_fonte;

**ATUALIZAR DADOS…….**

**UPDATE SET** [**WHERE**]: altera as linhas em uma tabela do banco de dados mas não a tabela como o alter table, tem que vir com o where , alteração também pode ser feita com base em uma coluna existente

ex:UPDATE nome\_da\_tabela

SET col1= “valor”

[WHERE condição];

ex:UPDATE cesta\_de\_frutas

SET preco\_unitario = 0.9 \* preco\_unitario

WHERE tipo\_da\_fruta IN ("banana", "maçã");

**REPLACE INTO VALUES:** Modifica valores, com base na chave primária da tabela, insira se não existir, e modifique com esse valores casoja exista

ex:REPLACE INTO nome\_da\_tabela

VALUES(valor1, valor2,....);

obs replace into é para colunas em tabelas que tem chave primária

**INSERT INTO VALUES ON DUPLICATE KEY UPDATE:** tenta inserir valores mas modifica caso a chave primária ja exista

ex:INSERT INTOnome\_da\_tabela (col1,col2,...)

VALUES(valor1, valor2,...)

ON DUPLICATE KEY UPDATE[expressao];

**DELETE FROM WHERE :** apaga um ou mais linhas em uma tabela do banco apartir de uma condição , diferente de drop table que delete a tabela e de truncate table que deleta todas as linhas da tabela

ex:DELETE FROM nome\_da\_tabela WHERE [condição]

**DELETE FROM WHERE ORDER BY LIMIT:**  aqui você pode ordenar a condição estabelecendo um limite até onde aquela condição na tabela é válida

DELETE FROM nome\_da\_tabela WHERE [condição]

ORDER BY coluna [LIMIT 1];

**PROCESSO “CRUD”:** CONSISTE em uma sequência de passos de operações básicas para atualizar dados ou fazer softwares interagir com o banco de dados, para o software interagir com o banco de dados, o software em questão irá precisar ter essas operações básicas:

#1-Create

cria a estrutura, tanto das tabelas quanto as dos dados e banco

ex: create database, create table , insert into tabela values(..);

#2- Read

ele vai lê, consultar a tabela/banco

ex: SELECT \* from tabela;

#3- Update

atualiza os dados do banco/tabela do sistema

ex:Alter table tabela, Update tabela, Replace Into tabela VALUES(..)

#4- Delete

Operação que apaga

Exemplos: Drop database bd, drop table tabela, truncate table tabela, delete from tabela (..)

**VARIÁVEIS**

**-VARIÁVEL INTEIRA:**

TinyInt, Smallint, Int, MediumInt, BigInt

**-VARIÁVEL REAL:**

Decimal,Float, Double, Real

**-VARIÁVEL LÓGICA:**

Bit, Boolean

**-VARIÁVEL DE DATA/TEMPO:**

Date, DateTime, TimeStamp, Time, Year

DATE: ano-mes-dia

DATETIME: ano-mes-dia horario

TIMESTAMP: quantidade de segundos que se passaram até agora após determinada data

-VARIÁVEL CARACTER**:**

Char, Varchar

-VARIÁVEL TEXTO**:**

TinyText, Text, MediumText, LongText

-VARIÁVEL BINÁRIO**:**

TinyBlob, Blob, MediumBlob, LongBlob

-VARIÁVEL COLEÇÃO:

Enum,Set

ENUM: Você pode adicionar vários objetos para aquela variável porém tem que adicionar e somente só adicionar um deles por linha

EXEMPLO :

**CREATE TABLE camisas (**

**idCamisa TINYINT PRIMARY KEY AUTO\_INCREMENT,**

**nome VARCHAR(25),**

**tamanho ENUM('pequena','média','grande','extra-grande')**

**);**

**INSERT INTO camisas (nome, tamanho)**

**VALUES ('regata', 'grande');**

**SELECT \* FROM camisas;**

SET: Você pode adicionar vários objetos para aquela variável e pode adicionar mais de um objeto na mesma linha ou nenhum

EXEMPLO:

**CREATE TABLE pedidos (**

**idPedido SMALLINT PRIMARY KEY AUTO\_INCREMENT,**

**lanche ENUM ('Presunto','Frango','Salame'),**

**tempero SET ('Azeite','Vinagre','Sal','Orégano','Pimenta')**

**);**

**INSERT INTO pedidos (lanche, tempero)**

**VALUES**

**('Presunto','Sal,Orégano'),**

**('Salame','Azeite,Pimenta'),**

**('Frango','Azeite,Sal,Orégano'),**

**('Presunto','');**

**SELECT \* FROM pedidos;**

VARIÁVEL ESPACIAL:

Geometry, Point, Polygon, MultiPolygon

**REGRAS UNIVERSAIS**

**1-**Quando a cardinalidade é M:N é necessário uma terceira tabela

no caso “M\_N”

**2-**Toda entidade tem que recebe um novo atributo codigo ou id\_ como chave primária

**3-** Atributos multivalorados(telefone) e atributos compostos(endereço) são separados como uma nova entidade cada uma

///

**TRANSAÇÃO**

**ACID**

**A-Atomicity:** atomicidade, trata a operação como algo único, indivisível, a transação deve ter todas as suas operações executadas em caso de sucesso. Nenhum resultado deve refletir na base de dados em caso de falha( sem resultados parciais

**C-Consistency:** consistência, toda operação deve respeitar as condições de consistência do banco de dados, isso significa considerar restrições como unidade de chaves, relações de integridade lógica, independente da lógica ou regras de negócio (da empresa) que sejam aplicadas

**I-Isolation:** isolamento, operações devem ser isoladas entre si, impedindo influência mútua. Operações exteriores a uma transação jamais têm acesso aos estados intermediários dessa operação, garante que o resultado de operações em paralelo seja o mesmo de que se essas operações fossem feitas em série. Enquanto as operações estão sendo executadas, todo o cenário intermediário dos dados não podem ser vistos por outras partes. O isolamento garante que várias ações podem ocorrer em paralelo ou seja ao mesmo tempo, ao em vez de um cliente inserir seus dados por vez todos podem inserir dados ao mesmo tempo

**D-Durability:** durabilidade, garante a durabilidade da informação, mesmo em um evento de pane total. Se o sistema toma a decisão de salvar as informações da transação, elas não podem ser mais perdidas. já vem por padrão nos softwares. sua informação é salva em caso de emergências, perda de energia elétrica e etc, o software sempre salva no intervalo de escrita

**PROBLEMA DO ACID:** A **performance** das operações realizadas é o maior potencial problema, sem as propriedades o desempenho pode ser melhor , mas a inconsistência torna o sistema proibitivo e improdutivo. Ainda assim, essa perda no desempenho pode tornar a aplicação inviável para algumas aplicações, claro, não todas, mas então o que fazer nesses casos ? Foi a partir desse problemas que surgiram outras aplicações como o NoSQL, ou não somente SQL( SQL e mais, banco de dados não relacional) o que irá envolver integração e conexão com outra plataformas, códigos, linguagens de programação, APIs e etc além de SQL, um sistema mais complexo enraizado sistematizado e robusto

**START TRANSACTION:** comando que marca o início da transação, todo código após ele é executado em regime intermediário

**COMMIT:** confirma e aplica ao banco de dados as alterações realizadas

**ROLLBACK:** utilizada em caso de erros , desfaz qualquer alteração realizada pela operação

ex:

START TRANSACTION;

COMMIT;

**SET:** Também é usado pra fazer comandos que alteram a configuração do SQL,

ex: SET autocommit=0;

INSERT INTO

COMMIT;

note que antes do commit os valores que você inserir no banco e as operações que você fizer só estarão disponíveis na sua conexão, caso você abra uma outra conexão, eles não estarão lá mas se você der um COMMIT , as modificações vão estar disponíveis em outras conexões

**–DECLARAÇÕES DE VÁRIAVEIS:** armazena valores consultaveis pelo SQL dentro da conexão ou sessão atual. São úteis para incrementar valores existentes no banco de dados, ou não precisar escrever várias vezes o mesmo valor. Idealmente são declardas antes de atribuitdas (não obrigatório em todos os softwares em SQL)

EX:

# declara a variável se usada dentro de uma transação, e não em um banco

**DECLARE** variavel\_1 **INT DEFAULT** 1;

#Atribuição da variável e declaração caso não usada dentro de uma transação, e sim em um banco

**SET** @variavel\_1 := 10;

#utilização em uma consulta

**SELECT \* FROM** sakila.customer

**WHERE** customer\_id <= @variavel\_1;

**–PROCEDIMENTOS ARMAZENADOS:** Trechos de código que executam uma ou mais operações específicas. Algumas vezes têm dentro de si uma transação (são conceitos diferentes!). Podem ter parâmetros de entrada ou saída(ou nenhum dos dois)

EX:

**DELIMITER** $$

**CREATE PROCEDURE** retorna\_clientes ()

**BEGIN**

**SELECT** first\_name, last\_name

**FROM** sakila.customer

**ORDER BY** first\_name;

**END**$$

**DELIMITER** ;

**CALL** retorna\_clientes; #tipo um select

NOTA: o call vai ser um select que não vai so retornar a tabela mas também pode retornar os valores dos parâmetros que você especificar entre parênteses

**EXEMPLO DE PROCEDIMENTO:**

CREATE DEFINER=`root`@`localhost` PROCEDURE `film\_in\_stock`(IN p\_film\_id INT, IN p\_store\_id INT, OUT p\_film\_count INT)

READS SQL DATA

BEGIN

SELECT inventory\_id

FROM inventory

WHERE film\_id = p\_film\_id

AND store\_id = p\_store\_id

AND inventory\_in\_stock(inventory\_id);

SELECT COUNT(\*)

FROM inventory

WHERE film\_id = p\_film\_id

AND store\_id = p\_store\_id

AND inventory\_in\_stock(inventory\_id)

INTO p\_film\_count;

END

**EXEMPLO DE CALL DO PROCEDIMENTO ANTERIOR**

CALL film\_in\_stock(30, 1, @variavel\_saida);

SELECT @variavel\_saida;

note: o @ é opcional, é apenas para executar, precisa colocar @ mas nada específico depois do @. e se você quiser selecionar o número de linhas é so logo depois do call fazer um select com a variável que tu declarou dentro do call-

**FUNCTIONS:** São que nem os procedimentos porém não usam o CALL , e sim select

**EXEMPLO DE FUNCTIONS:**

CREATE DEFINER=`root`@`localhost` FUNCTION `inventory\_in\_stock`(p\_inventory\_id INT) RETURNS tinyint(1)

READS SQL DATA

BEGIN

DECLARE v\_rentals INT;

DECLARE v\_out INT;

#AN ITEM IS IN-STOCK IF THERE ARE EITHER NO ROWS IN THE rental TABLE

#FOR THE ITEM OR ALL ROWS HAVE return\_date POPULATED

SELECT COUNT(\*) INTO v\_rentals

FROM rental

WHERE inventory\_id = p\_inventory\_id;

IF v\_rentals = 0 THEN

RETURN TRUE;

END IF;

SELECT COUNT(rental\_id) INTO v\_out

FROM inventory LEFT JOIN rental USING(inventory\_id)

WHERE inventory.inventory\_id = p\_inventory\_id

AND rental.return\_date IS NULL;

IF v\_out > 0 THEN

RETURN FALSE;

ELSE

RETURN TRUE;

END IF;

END

**CONSULTANDO VALORES DO PROCEDIMENTO ANTERIOR:**

SELECT inventory\_in\_stock(34)

Note ele retornar a quantidade em stock no valor/parâmetro 34

**OBSERVAÇÃO:** para saber consultar valores de procedimentos encontre sempre as variáveis declaradas:

exemplo generico de call e select em procedimentos

CALL procedure(‘’,’’’,’’)

SELECT procedimento(‘’,’’) ou SELECT procedimento (‘’)

**HIERARQUIA DA ARQUITETURA DE UM BANCO DE DADOS**

**SELECT \* FROM mysql.user; :** no mysql esse comando retorna a tabela que informa todos os usuários que podem acessar, mostra todos os usuarios

**ROOT:** usuario padrão do banco de dados, com senha configurada na instação. esse acesso tem todas as permissões no banco de dados e pode fazer qualquer coisa. no computador pessoal não há problema mas em um ambiente real pode ser uma vulnerabilidade

**CREATE USER** ‘’ ’’’ **IDENTIFIED BY** ‘‘ ’’; Cria um novo usuário de acesso com senha ao banco de dados. Nessa etapa também é possível determinar limitações ou vencimento para a senha. Tomar cuidado com senha padrão para não diminuir a segurança do banco de dados

exemplo: CREATE USER 'felippe'@'localhost' IDENTIFIED BY '123456'

**CREATE USER** ‘’ ’’’ **COMMENT** ‘‘ ’’; cria usuário com comentário

**SELECT CURRENT\_USER ();** retorna o usuário atual

**ALTER USER** ‘’ ’’ **ATTRIBUTE;** adiciona atributo ao usuário  
  
**ALTER USER** ‘’ ’’ **COMMENT:**  alterar comentário de usuário

**DROP USER** ‘felipe’ @ ‘localhost’ ; deleta usuário do host

**DROP USER** ‘felipe’ @ ‘%’; deleta o usuário em todo o banco

**PRIVILÉGIOS:** permissão para que determinado usuário (ou conjunto) execute algum comando. São privilégios que determinam quais operações podem ser realizadas. variam nas mais diversas granularidade e tipos de controle e permissionamento

**GRANT SELECT ON** ‘banco’ **TO** ‘user’ **:** atribui uma permissão a um usuário

**PAPÉIS:** grupo de privilégios que facilitam o gerenciamento de permissões. podem existir privilégios para engenharia, visualização e administração por exemplo. podem ser criados, alterados e deletados com os mesmos comandos que usuários

**CREATE ROLE** “nome do papel”: cria um papel

**exemplo:**

**# cria o papel chamado engenharia**

CREATE ROLE 'engenharia'@'localhost';

**#permissão para o usuário consultar e inserir dados em qualquer banco de dados do root, ou seja do usuário padrão do mysql**

GRANT SELECT, INSERT ON \*.\* TO 'engenharia'@'localhost';

**# associar o papel para felipe**

GRANT 'engenharia'@'localhost' TO 'felippe'@'localhost';

**obs:**

**SET GLOBAL activate\_all\_roles\_on\_login = TRUE;** é importante ativar esse comando, assim vai configurar seu software para permitir manipulação de papéis e privilégios , ou seja vc consegue manipular o controle de acesso, usar tais ferramentas

**Grant :** Dá acesso

**Revoke:** revoga acesso

**Informações da tua conta Trackmania Forever:**

**O teu nome de início de sessão: old\_edinho**

**A tua palavra-passe: filho99887**

**A tua chave: TMNF-FKTC-FRNW-47X5-P5M**