SQL-Structured Query Language

Quando utilizado o **SELECT** statement, o sistema da base de dados valida **primeiro** o **FROM** e depois o **SELECT**

“Da tabela, seleciona data destas colunas”

SQL Query Básico

```
SELECT
  id
FROM
  Users;
```

Fazer cálculos SQL

```
SELECT
  salario * 1.05
FROM
  empregados;
```

Isto adiciona 5% ao salário dos empregados, vai criar uma coluna com o nome “salario * 1.05” para alterar basta só meter á frente da conta AS novo_salario.

```
SELECT
  salario * 1.05 AS novo_salario
FROM
  empregados;
```

SQL ORDER BY

O `ORDER BY` é uma cláusula opcional do `SELECT` statement. O `ORDER BY` deixa ordenar por crescente ou decrescente o statement do `SELECT`

```
SELECT
  lista
FROM
  tabela
ORDER BY
  coluna [ASC | DESC];
```

Filtrar DATA

DISTINCT

Para remover linhas duplicadas de um resultado, usa-se o operador `DISTINCT` na cláusula `SELECT`.

```
SELECT DISTINCT
    column1, column2, ...
FROM
    table1;
```

Nota: O DISTINCT só apaga as linhas duplicadas de um resultado, não apaga tabelas.

Para uma tabela

```
SELECT
  DISTINCT
  salario
FROM
  empregados
ORDER BY
  job_id,
  salario DESC;
```

Para várias tabelas

```
SELECT DISTINCT
  salario,
  job_id
FROM
  empregados
ORDER BY
  job_id,
  salario DESC;
```

LIMIT

Para limitar o número de linhas retornadas por um `SELECT`, utiliza-se as cláusulas `LIMIT` e o `OFFSET`.

Modo de utilização

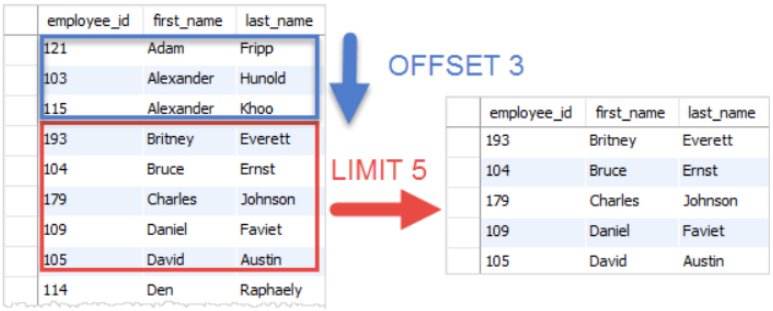
```
SELECT
    column_list
FROM
```

```
    table1
ORDER BY column_list
LIMIT row_count OFFSET offset;
```

- O `LIMIT row_count` determina o número de linhas (`row_count`) retornadas pela query.
- A cláusula `OFFSET offset` faz skip à linha `offset` antes de retornar as linhas.

O `OFFSET` é opcional, quando se utiliza o `LIMIT` é **importante** usar o `ORDER BY` para assegurar que está organizado.

LIMIT **não** é suportado por todas as bases de dados



FETCH

Como o `LIMIT` **não** é suportado por todas as bases de dados existe o `OFFSET FETCH`. Ele permite pular `N` linhas de um resultado antes de começar a retornar alguma linha.

Syntax utilizada pelo FETCH

```
OFFSET offset_rows { ROW | ROWS }
FETCH { FIRST | NEXT } [ fetch_rows ] { ROW | ROWS } ONLY
```

Exemplo

```
SELECT
    employee_id,
    first_name,
    last_name,
    salary
FROM employees
ORDER BY
    salary DESC
OFFSET 0 ROWS
FETCH NEXT 1 ROWS ONLY;
```

WHERE

Para especificar uma linha de uma tabela utiliza-se o `WHERE` com o `SELECT` statement. Possível com o `UPDATE` e o `DELETE` para especificar as linhas para serem atualizadas ou apagadas.

```
SELECT
    column1, column2, ...
FROM
    table_name
WHERE
    condition;
```

Operadores de Comparação

Operator	Meaning
=	Equal
<>	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Operadores Lógicos

Operator	Meaning
ALL	Return true if all comparisons are true
AND	Return true if both expressions are true
ANY	Return true if any one of the comparisons is true.
BETWEEN	Return true if the operand is within a range
EXISTS	Return true if a subquery contains any rows
IN	Return true if the operand is equal to one of the value in a list
LIKE	Return true if the operand matches a pattern
NOT	Reverse the result of any other Boolean operator.
OR	Return true if either expression is true
SOME	Return true if some of the expressions are true

Wild Card

```
SELECT
    employee_id, first_name, last_name
FROM
    employees
WHERE
    first_name LIKE 'jo%'
ORDER BY first_name;
```

Mostra os nomes com `jo` no inicio da palavra, para fazer o inverso basta meter o sinal no outro lado `%jo`.

O `%` representa 0, 1 ou vários caracteres.

O `_` representa 1 único caracter.

CASE Expression

A expressão `CASE` permite validar uma lista de condições e retornar um dos possíveis resultados. O CASE tem dois formatos, simple `CASE` e o searched `CASE`.

Pode-se utilizar numa cláusula ou statement que permita uma expressão válida. Por exemplo, usar o CASE com `SELECT`, `DELETE` e o `UPDATE` ou cláusulas como `SELECT`, `ORDER BY`, `HAVING`.

```
CASE expression
WHEN when_expression_1 THEN
  result_1
WHEN when_expression_2 THEN
  result_2
WHEN when_expression_3 THEN
  result_3
...
ELSE
  else_result
END
```

Joining Multiple Tables

Column Aliases

Quando criamos a base de dados é normal usar siglas para manter as palavras mais simples.

- O `so_no` pode ser sales order number
- O `qty` pode ser quantidade

É possível dar nomes temporários às colunas, utilizando o `AS`

`nome_coluna AS nova_coluna`

Caso tenha expressões utiliza-se a mesma técnica

```
SELECT
    first_name,
    last_name,
    salary * 1.1 AS new_salary
FROM
    employees;
```

Erro Comum

```
SELECT
    first_name,
    last_name,
    salary * 1.1 AS new_salary
FROM
    employees
WHERE new_salary > 5000
```

Dá erro porque a base de dados faz isto primeiro

`FROM > WHERE > SELECT`

Ou seja dá erro porque faz o `WHERE` primeiro com o nome novo..

Para resolver isto utiliza-se o `ORDER BY`

```
SELECT
    first_name,
    last_name,
    salary * 1.1 AS new_salary
FROM
    employees
ORDER BY new_salary;
```

Desta maneira a base de dados vai fazer a seguinte ordem

`FROM > SELECT > ORDER BY`

Table Aliases

Para trocar o nome da tabela faz se o seguinte comando

`nome_tabela AS nova_tabela`

Nota: O nome da tabela não vai ser renomeado completamente, é temporário.

Então porque utilizar aliases?

Quando especificamos nomes de colunas com a cláusula SELECT podemos utilizar a seguinte syntax.

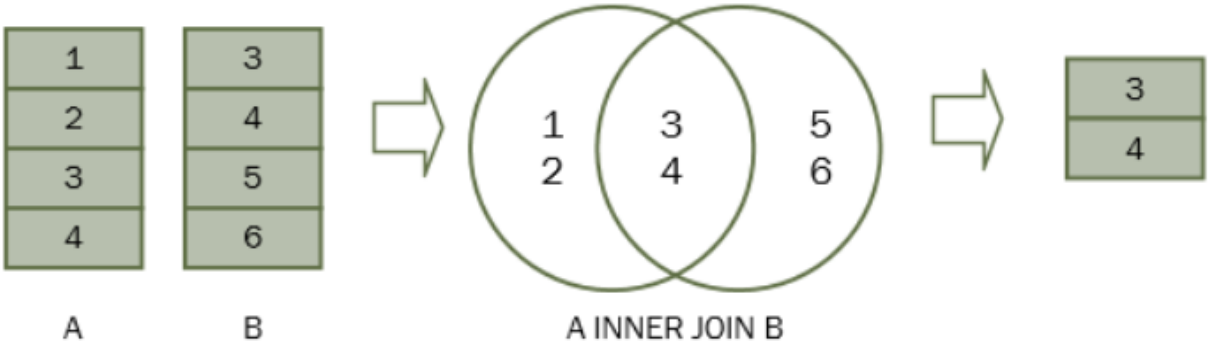
`table_name.column_name`

Por exemplo:

```
SELECT
    employees.first_name,
    employees.last_name
FROM
    employees;
```

Inner Join

Serve para connectar duas tabelas diferentes com um resultado que seja o mesmo em ambas as tabelas.



Exemplo

```
SELECT coluna_a
FROM A
INNER JOIN B ON coluna_b = coluna_a;
```

Outro exemplo

```
SELECT
    first_name,
    last_name,
    employees.department_id,
    departments.department_id,
    department_name
FROM
    employees
        INNER JOIN
    departments ON departments.department_id = employees.department_id
WHERE
    employees.department_id IN (1 , 2, 3);
```

Aggregate Functions

É usualmente utilizado pela cláusula `GROUP BY` do statement `SELECT`. O `GROUP BY` divide o resultado em grupos de valores e a função aggregate retorna o valor único de cada grupo.

O seguinte código mostra como usar a função aggregate

```
SELECT c1, aggregate_function(c2)
FROM table
GROUP BY c1;
```

- AVG() - retorna a média de um conjunto
- COUNT() - retorna o número de um conjunto
- MAX() - retorna o valor máximo
- MIN() - retorna o valor mínimo
- SUM() - retorna a soma de todos os valores de um conjunto

Exepto a função COUNT(), as funções de aggregate **ignoram** o NULL.

https://www.sqltutorial.org/sql-aggregate-functions/

AVG

O `AVG()` retorna a média de valores num grupo.

```
AVG( ALL | DISTINCT)
```

O próximo exemplo mostra a média do salário de cada departamento.

```
SELECT
    department_name, ROUND(AVG(salary), 0) avg_salary
FROM
    employees
        INNER JOIN
```

```
    departments USING (department_id)
GROUP BY department_name
ORDER BY department_name;
```

MIN

O `MIN()` retorna o valor mínimo de um grupo.

```
MIN(column | expression)
```

O próximo exemplo mostra o valor mínimo do salário de cada trabalhador.

```
SELECT
    department_name, MIN(salary) min_salary
FROM
    employees
        INNER JOIN
    departments USING (department_id)
GROUP BY department_name
ORDER BY department_name;
```

MAX

o MAX() retorna o valor máximo de um grupo.

```
MAX(column | expression)
```

O próximo exemplo mostra o valor máximo do salário dos trabalhadores de cada departamento.

```
SELECT
    department_name, MAX(salary) highest_salary
FROM
    employees
        INNER JOIN
    departments USING (department_id)
GROUP BY department_name
ORDER BY department_name;
```

COUNT

O COUNT() retorna o número de itens de um grupo.

```
COUNT ( [ALL | DISTINCT] column | expression | *)
```

O próximo exemplo mostra os números de funcionários de cada departamento.

```
SELECT
    department_name, COUNT(*) headcount
FROM
    employees
        INNER JOIN
    departments USING (department_id)
GROUP BY department_name
ORDER BY department_name;
```

SUM

O SUM() retorna a soma de todos os valores de um grupo.

```
SUM(ALL | DISTINCT column)
```

O próximo exemplo mostra o total do salário de cada departamento.

```
SELECT
    department_id, SUM(salary)
FROM
    employees
GROUP BY department_id;
```

Grouping Data

GROUP BY

Para agrupar linhas em grupos, utiliza-se o `GROUP BY`.

É muito utilizado com as aggregate functions.

Exemplo

```
SELECT
  column1,
  column2,
  AGGREGATE_FUNCTION (column3)
FROM
  table1
GROUP BY
  column1,
  column2;
```

HAVING

Para especificar uma condição após o `GROUP BY` geralmente é utilizado o `HAVING`.

Exemplo

```
SELECT
  column1,
  column2,
  AGGREGATE_FUNCTION (column3)
FROM
  table1
GROUP BY
  column1,
  column2
HAVING
  group_condition;
```

Nota: A cláusula `HAVING` aparece logo asseguir ao `GROUP BY`.

HAVING vs WHERE

O `WHERE` aplica a condição antes do `GROUP BY` e o `HAVING` faz a condição depois das linhas estarem agrupadas.

Set Operators

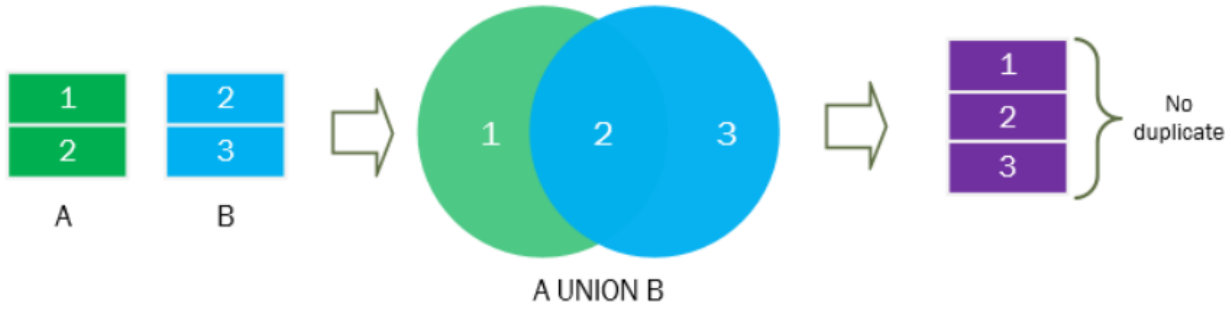
UNION

O `UNION` junta uma query à proxima.

Exemplo

```
SELECT
    column1, column2
FROM
    table1
UNION
SELECT
```

```
    column3, column4
FROM
    table2;
```

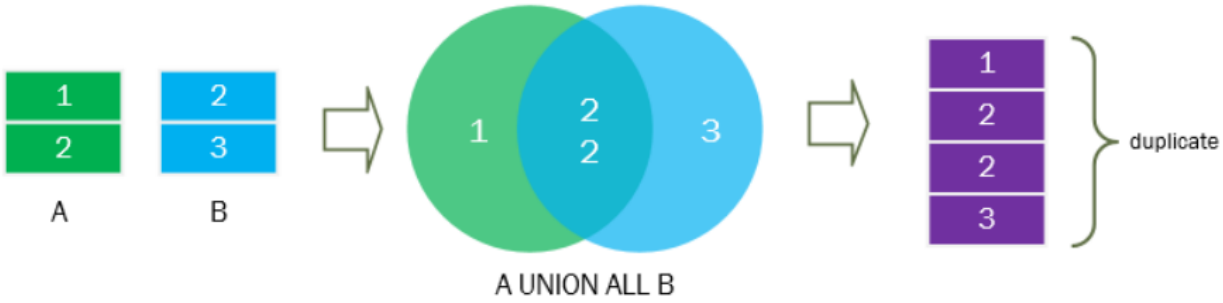


UNION ALL

O `UNION ALL` ele junta mas **duplica**, **acrescenta** os dados que forem iguais.

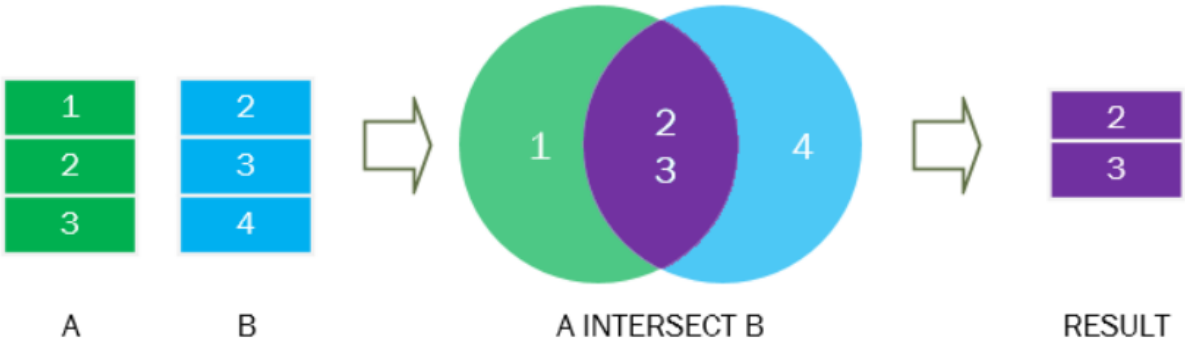
Exemplo

```
SELECT
    column1, column2
FROM
    table1
UNION ALL
SELECT
    column3, column4
FROM
    table2;
```



INTERSECT

O `INTERSECT` só mostra as linhas que são **iguais, os dados que são intersetados**.



Exemplo

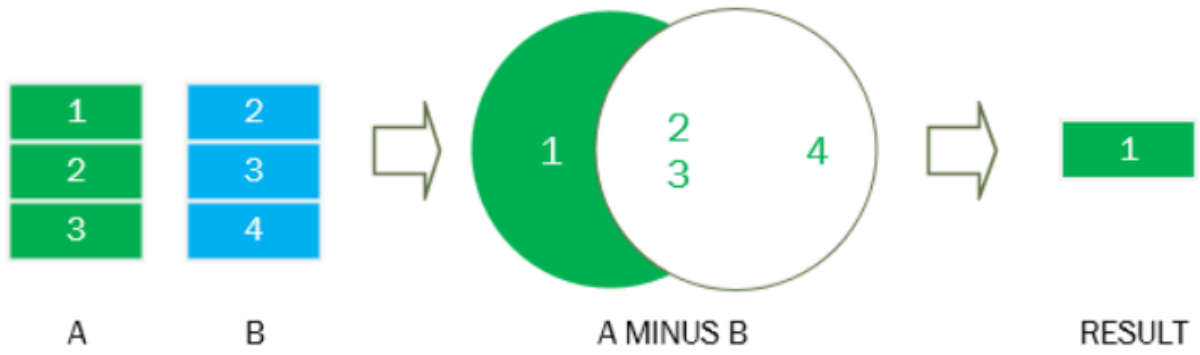
```
SELECT
  id
FROM
  a
INTERSECT
SELECT
  id
FROM
  b;
```

MINUS

Fora os operadores `UNION`, `UNION ALL` e o `INTERSECT` existe o `MINUS` que permite subtrair um resultado a outro.

Exemplo

```
SELECT
  id
FROM
  A
MINUS
SELECT
  id
FROM
  B;
```



Subquery

A subquery permite fazer queries dentro de uma query.

Exemplo

```
SELECT
    employee_id, first_name, last_name
FROM
    employees
WHERE
    department_id IN (SELECT
            department_id
        FROM
            departments
        WHERE
            location_id = 1700)
ORDER BY first_name , last_name;
```

É **permitido** utilizar este método nas seguintes situações.

- Com os operadores `IN` ou `NOT IN`
- Com os operadores de comparação
- Com os operadores `EXISTS` ou `NOT EXISTS`
- Com os operadores `ANY` ou `ALL`
- Na clausula `FROM`
- Na clausula `SELECT`

Mais Exemplos

`IN` ou `NOT IN`

```
SELECT
    employee_id, first_name, last_name
FROM
    employees
WHERE
    department_id NOT IN (SELECT
            department_id
        FROM
            departments
        WHERE
            location_id = 1700)
ORDER BY first_name , last_name;
```

Com os Operadores de comparação

```
SELECT
    employee_id, first_name, last_name, salary
FROM
    employees
WHERE
```

```
    salary = (SELECT
            MAX(salary)
        FROM
            employees)
ORDER BY first_name , last_name;
```

Com os operadores `EXISTS` ou `NOT EXISTS`

```
SELECT
    department_name
FROM
    departments d
WHERE
    EXISTS( SELECT
            1
        FROM
            employees e
        WHERE
            salary > 10000
                AND e.department_id = d.department_id)
ORDER BY department_name;
```

Correlated Subquery

A correlated subquery é práticamente utilizadar uma subquery que utilize outra subquery.

Exemplo

```
SELECT
    employee_id,
    first_name,
    last_name,
    salary,
    department_id
FROM
    employees e
WHERE
    salary > (SELECT
            AVG(salary)
        FROM
```

```
            employees
        WHERE
            department_id = e.department_id)
ORDER BY
    department_id ,
    first_name ,
    last_name;
```

Correlated Subquery com EXISTS

O operador `EXISTS` é bastante utilizado.

Exemplo

```
SELECT
    employee_id,
    first_name,
    last_name
FROM
    employees e
WHERE
    NOT EXISTS( SELECT
            *
        FROM
            dependents d
        WHERE
            d.employee_id = e.employee_id)
ORDER BY first_name ,
         last_name;
```

EXISTS

O Operador `EXISTS` permite fazer uma subquery para testar a existencia de linhas.

```
EXISTS (subquery)
```

Exemplo

```
SELECT
    employee_id, first_name, last_name
FROM
    employees
WHERE
    EXISTS( SELECT
            1
        FROM
            dependents
        WHERE
            dependents.employee_id = employees.employee_id);
```

Com o do `NOT EXISTS`

```
SELECT
    employee_id, first_name, last_name
FROM
    employees
WHERE
    NOT EXISTS( SELECT
            1
        FROM
            dependents
        WHERE
            dependents.employee_id = employees.employee_id);
```

ALL

O Operador `ALL` compara valores únicos com uma coluna única de valores.

```
WHERE column_name comparison_operator ALL (subquery)
```

Condições

Condition	Meaning
c > ALL(...)	The values in column c must greater than the biggest value in the set to evaluate to true.
c >= ALL(...)	The values in column c must greater than or equal to the biggest value in the set to evaluate to true.
c < ALL(...)	The values in column c must be less than the lowest value in the set to evaluate to true.
c >= ALL(...)	The values in column c must be less than or equal to the lowest value in the set to evaluate to true.
c <> ALL(...)	The values in column c must not be equal to any value in the set to evaluate to true.
c = ALL(...)	The values in column c must be equal to any value in the set to evaluate to true.

Exemplo

```
SELECT
    *
FROM
    table_name
WHERE
    column_name > ALL (subquery);
```

ANY

O Operador `ANY` compara um grupo de valores com o resultado returnado da subquery.

```
WHERE column_name comparison_operator ANY (subquery)
```

Condição

Condition	Meaning
x = ANY (...)	The values in column c must match one or more values in the set to evaluate to true.
x != ANY (...)	The values in column c must not match one or more values in the set to evaluate to true.
x > ANY (...)	The values in column c must be greater than the smallest value in the set to evaluate to true.
x < ANY (...)	The values in column c must be smaller than the biggest value in the set to evaluate to true.
x >= ANY (...)	The values in column c must be greater than or equal to the smallest value in the set to evaluate to true.
x <= ANY (...)	The values in column c must be smaller than or equal to the biggest value in the set to evaluate to true.

Exemplo

```
SELECT
    first_name,
    last_name,
    salary
FROM
    employees
WHERE
    salary = ANY (
        SELECT
            AVG(salary)
        FROM
            employees
        GROUP BY
            department_id)
ORDER BY
    first_name,
```

```
    last_name,
    salary;
```

Modifying Data

INSERT

O `INSERT` possibilita adicionar linhas às tabelas, ele permite fazer as seguintes coisas.

- Inserir uma linha única numa tabela.
- Inserir várias linhas numa tabela.
- Copia linhas de uma tabela para outra.

```
INSERT INTO table1 (column1, column2,...)
VALUES
  (value1, value2,...);
```

Exemplo

```
INSERT INTO dependents (
  first_name,
  last_name,
  relationship,
  employee_id
)
VALUES
  (
    'Dustin',
    'Johnson',
    'Child',
    178
  );
```

UPDATE

O UPDATE é utilizado para alterar data numa tabela.

```
UPDATE table_name
SET column1 = value1,
 column2 = value2
WHERE
  condition;
```

Exemplo

```
UPDATE employees
SET
    last_name = 'Lopez'
WHERE
    employee_id = 192;
```

DELETE

O DELETE é utilizado para apagar linhas únicas ou várias de uma tabela.

```
DELETE
FROM
  table_name
WHERE
  condition;
```

Exemplo

```
DELETE FROM dependents
WHERE
    dependent_id = 16;
```

Working whit table structures

CREATE TABLE

O CREATE TABLE serve para criar uma tabela.

```
CREATE TABLE table_name(
     column_name_1 data_type default value column_constraint,
     column_name_2 data_type default value column_constraint,
     ...,
     table_constraint
);
```

Exemplo

```
CREATE TABLE courses (
    course_id INT AUTO_INCREMENT PRIMARY KEY,
    course_name VARCHAR(50) NOT NULL
);
```

Outro exemplo

```
CREATE TABLE trainings (
    employee_id INT,
    course_id INT,
    taken_date DATE,
    PRIMARY KEY (employee_id , course_id)
);
```

ALTER TABLE

O `ALTER TABLE` serve para modificar a estrutura da tabela. É possivel fazer as seguintes coisas.

- Adicionar uma coluna nova utilizando a clausula `ADD`.
- Modificar o atributo de uma coluna utilizando o `MODIFY`.
- Remover tabelas utilizando o `DROP`.

Adicionar uma tabela com ALTER TABLE

```
ALTER TABLE table_name
ADD new_colum data_type column_constraint [AFTER existing_column];
```

Exemplo

```
ALTER TABLE courses
ADD fee NUMERIC (10, 2) AFTER course_name,
ADD max_limit INT AFTER course_name;
```

Modificar uma tabela com ALTER TABLE

```
ALTER TABLE table_name
MODIFY column_definition;
```

Exemplo

```
ALTER TABLE courses
MODIFY fee NUMERIC (10, 2) NOT NULL;
```

Apagar colunas com ALTER TABLE

```
ALTER TABLE table_name
DROP column_name,
DROP colum_name,
...
```

Exemplo

```
ALTER TABLE courses
DROP COLUMN max_limit,
DROP COLUMN credit_hours;
```

DROP TABLE

O `DROP TABLE` é utilizado para apagar tabelas.

```
DROP TABLE [IF EXISTS] table_name;
```

Exemplo

```
DROP TABLE emergency_contacts;
```

Exemplo remover várias tabelas

```
DROP TABLE table_name1,table_name2,...;
```

TRUNCATE TABLE

O TRUCANTE TABLE é utilizado para apagar tabelas com **muita** informação.

```
TRUNCATE TABLE table_name;
```

Algumas bases de dados não suporta essa syntax.

```
TRUNCATE table_name;
```

Várias tabelas.

```
TRUNCATE TABLE table_name1, table_name2, ...;
```

TRUNCATE TABLE vs DELETE

- Com o `DELETE` é possivel voltar a tráz porque guarda logs, como TRUCATE não dá para recuperar a informação.
- Para apagar uma tabela que tenha uma KEY não conhecida, não é possível utilizar o `TRUNCATE`.

Constraints

Primary Key

A Primary Key, serve para identificar colunas, cada tabela tem **uma e uma só** Primary Key.

Criar uma tabela com Primary Key.

```
CREATE TABLE projects (
    project_id INT PRIMARY KEY,
    project_name VARCHAR(255),
    start_date DATE NOT NULL,
    end_date DATE NOT NULL
);
```

Utilizando o ALTER TABLE

```
ALTER TABLE project_milestones
ADD CONSTRAINT pk_milestone_id PRIMARY KEY (milestone_id);
```

Remover uma Primary Key com ALTER TABLE

```
ALTER TABLE table_name
DROP CONSTRAINT primary_key_constraint;
```

Com outras bases de dados (MySQL)

```
ALTER TABLE table_name
DROP PRIMARY KEY;
```

Foreign Key

A Foreign Key é uma coluna ou grupo de colunas que faz uma ligação entre duas tabelas, a coluna Primary Key da primeira tabela é referenciada pela coluna da segunda tabela. A coluna da segunda passa a ser a Foreign Key.

Adicionar uma Foreign Key a tabelas existentes

```
ALTER TABLE table_1
ADD CONSTRAINT fk_name FOREIGN KEY (fk_key_column)
   REFERENCES table_2(pk_key_column)
```

Para defenir uma Foreign Key

```
ALTER TABLE project_milestones
ADD CONSTRAINT fk_project FOREIGN KEY(project_id)
   REFERENCES projects(project_id);
```

Remover uma Foreign Key

```
ALTER TABLE table_name
DROP CONSTRAINT fk_name;
```

Com MySQL

```
ALTER TABLE table_name
DROP FOREIGN KEY fk_name;
```

UNIQUE

De vez em quando é necessário garantir que certos valores **não** sejam duplicados.

A coluna do email não é uma Primary Key, e **não** pode ser um valor igual, para isso utiliza-se o UNIQUE.

UNIQUE vs Primary Key

	PRIMARY KEY constraint	UNIQUE constraint
The number of constraints	One	Many
NULL values	Do not allow	Allow

Criar com UNIQUE

```
CREATE TABLE users (
    user_id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(255) NOT NULL UNIQUE,
    password VARCHAR(255) NOT NULL
);
```

Adicionar UNIQUE a uma tabela existente

```
ALTER TABLE users
ADD CONSTRAINT uc_username UNIQUE(username);
```

Criar uma nova coluna

```
ALTER TABLE users
ADD new_column data_type UNIQUE;
```

Exemplo

```
ALTER TABLE users
ADD email VARCHAR(255) UNIQUE;
```

Remover UNIQUE

```
ALTER TABLE table_name
DROP CONSTRAINT unique_constraint_name;
```

Exemplo

```
ALTER TABLE users
DROP CONSTRAINT uc_username;
```

NOT NULL

O `NOT NULL` serve para indicar que uma coluna **não** pode ter valores nulos.

Isso indica que quando utiliza-mos o `INSERT` temos que especificar os valores para as colunas `NOT NULL`.

```
CREATE TABLE table_name(
   ...
   column_name data_type NOT NULL,
   ...
);
```

Exemplo

```
CREATE TABLE training (
    employee_id INT,
    course_id INT,
    taken_date DATE NOT NULL,
    PRIMARY KEY (employee_id , course_id)
);
```

Com INSERT

```
INSERT INTO training(employee_id,course_id)
VALUES(1,1);
```

ALTER TABLE NOT NULL

As vezes quando criamos as tabelas esquecemos de utilizar o `NOT NULL`, para conseguir alterar depois da criação da tabela utiliza-se o `ALTER TABLE`.

Primeiro altera-se os valores `NULL` para `NOT NULL`

```
UPDATE table_name
SET column_name = 0
WHERE
  column_name IS NULL;
```

Adicionar o NOT NULL

```
ALTER TABLE table_name
MODIFY column_name data_type NOT NULL;
```

Exemplo

```
UPDATE training
SET taken_date = CURRENT_DATE ()
WHERE
  taken_date IS NULL;
```

Alterar para NOT NULL

```
ALTER TABLE training
MODIFY taken_date date NOT NULL;
```

CHECK

O `CHECK` faz com que uma coluna tenha valores booleanos (True,False)

```
CONSTRAINT constraint_name CHECK(Boolean_expression)
```

Exemplos

```
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(255) NOT NULL,
    selling_price NUMERIC(10,2) CHECK (selling_price > 0)
);
```