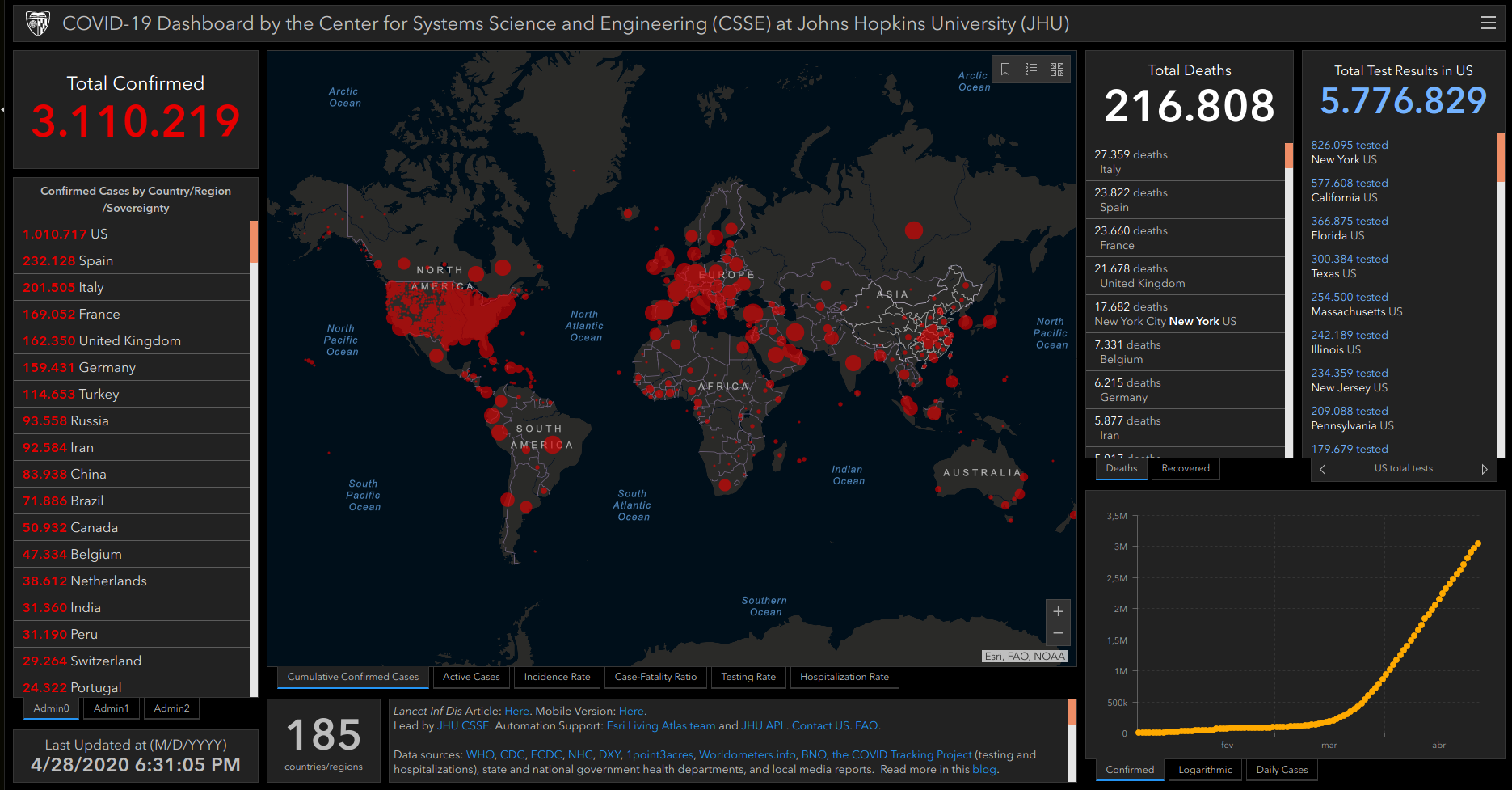
# Onde vivem os dados no mundo real?

Pense na quantidade imensa de dados que você fornece a diferentes empresas e instituições diariamente. Quando acessa seu email, onde estão guardadas suas mensagens e como o site "confere" suas credenciais para lhe permitir acessá-los? Em um jornal online, onde são salvas as matérias? O que permite aos caixas eletrônicos e ao sistema financeiro de maneira geral ser tão seguros e garantirem que dois saques quase simultâneos não sejam contabilizados, para sua infelicidade, como apenas um único?

Se você está do outro lado do balcão, numa empresa, provavelmente gostaria de responder a questões pertinentes de negócio de forma mais sólida em termos quantitativos. Ouvir um elogio da Maria e do Paulo, clientes antigos, ou uma sugestão de melhoria do Pedro, certamente pode levar a insights úteis, mas também conclusões enviesadas: Maria, Paulo e Pedro são representativos do universo de clientes? Aliás, quem são os clientes? Quais seus padrões de compras? Como o último trimestre de vendas nas três maiores cidades atendidas pela empresa se compara ao mesmo período do ano passado?

É claro que para responder a essas perguntas precisamos gráficos e estatística. Mas, antes de tudo, precisamos organizar esses dados de maneira inteligente para podermos inserir, atualizar, apagar e, claro, consultar esses registros. Os bancos de dados são os softwares especializados nisso.



Dashboard de referência para acompanhamento da propagação do coronavírus em tempo real. Em geral, dashboards assim operam consultando continuamente um banco de dados

## Tipos de bancos de dados

Hoje em dia existem diversos tipos de bancos de dados, refletindo casos de uso cada vez mais específicos. Podemos dividi-los em dois grandes grupos: relacionais e não-relacionais. Grosso modo, a diferença principal entre os dois é que os primeiros armazenam os dados em diversas tabelas interconectadas, de forma que lembra bastante as abas numa planilha do Excel, enquanto os segundos não: há bancos não-relacionais que salvam os dados como JSONs, pares chave-valor, grafos, etc.

Embora bancos não-relacionais tenham cada vez mais importância por conta do Big Data e das necessidades de prototipação rápida, saber consultar bancos de dados relacionais é mandatório na maioria das empresas de tecnologia para posições de analistas e engenheiros e este curso dará ênfase neste último.

### Exemplos de bancos relacionais:

* Postgres
* MySQL
* SQL Server
* SQLite

### Exemplos de bancos não-relacionais:

* MongoDB
* Cassandra
* Redis
* Neo4j

Neste curso, focaremos no Postgres. Na seção de bancos não-relacionais, trabalharemos com o MongoDB.

## A língua dos bancos de dados

A linguagem para manter e consultar bancos relacionais se chama SQL (Structured Query Language). Esta não é uma liguagem de propósito geral, como Python ou Java, mas uma feita para operar bases de dados em bancos relacionais. Em contraposição, usa-se inclusive o termo NoSQL (Not Only SQL) para se referir a bancos de dados não-relacionais.

Ok, mas qual a cara desses códigos, afinal? Hipoteticamente, se tivéssemos nossos dados históricos de vendas num banco relacional, o código SQL para obter todas as vendas de janeiro seria:

SELECT \*

FROM vendas

WHERE mes = 'janeiro';

Já num banco como o MongoDB, não-relacional, a mesma consulta teria a seguinte forma:

db.vendas.find({

"mes": "janeiro"

});

Vamos explorar os detalhes de cada sintaxe nas próximas aulas.

Como uma língua, o SQL possui "dialetos" levemente diferentes entre si, dependendo de qual banco relacional (ou "banco SQL") você esteja usando. Assim, não estranhe se seu código de consulta, ou query, escrito numa base MySQL não funcionar direito num banco Postgres.

## Todo mundo gosta de SQL

Vale dizer que, por conta da onipresença dos bancos relacionais, diversos sistemas de armazenamento tentam manter uma sintaxe semelhante ao do SQL, embora estejam processando as coisas por baixo dos panos de forma muito diferente.

O Hive, um sistema para processamento de big data, por exemplo, possui uma sintaxe SQL de consulta, embora esteja traduzindo seus códigos para Java de maneiras bem complicadas sem que você perceba.

No caso do MongoDB, sua sintaxe é completamente diferente do SQL. Isto não impede, no entanto, que tentativas de criar tradutores surjam. Este aqui é um bom exemplo: [www.querymongo.com](http://www.querymongo.com/) - a experiência prática lhe mostrará suas limitações...

# First things first

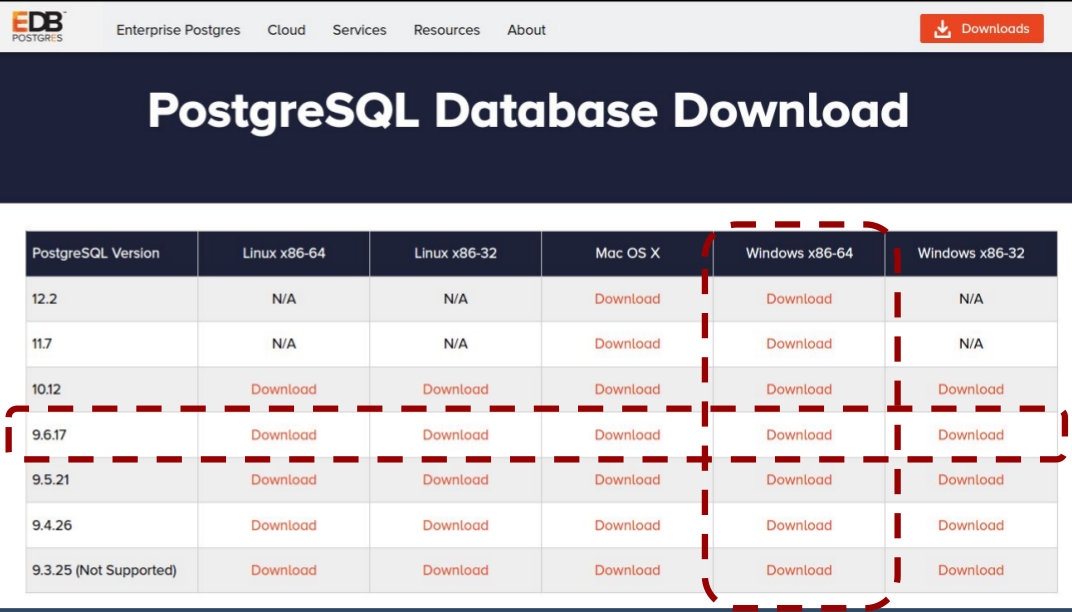
Vamos começar instalando um banco local - isto é, no seu próprio computador. Tudo começa baixando o servidor. Como mencionado na Introdução, neste curso usaremos o Postgres como exemplo de banco relacional, porque ele é gratuito e bastante utilizado - o Redshift, o data warehouse na nuvem da Amazon e um dos bancos analíticos mais usados no mundo, inclusive, é baseado no próprio Postgres; assim, muito dos princípios e da sintaxe são compartilhados.

## Instalando o servidor

É possível baixar PostgreSQL Server da página de downloads do PostgreSQL: <https://www.postgresql.org/download/>

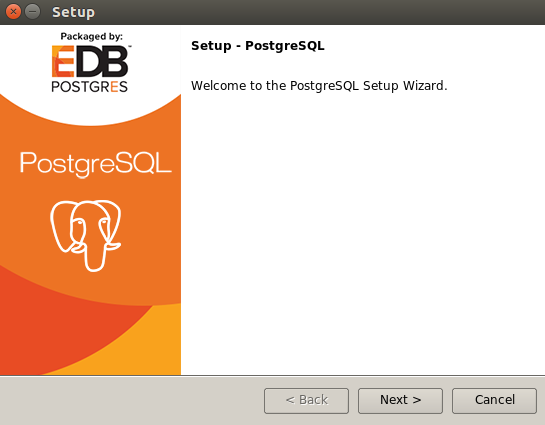
Para simplificar, vamos baixar os instaladores direto deste link: <https://www.enterprisedb.com/downloads/postgres-postgresql-downloads>

Na tabela que surgir, vamos baixar a versão 9.6.\*. Baixe o instalador para o seu Sistema Operacional - em geral, apenas computadores antigos são 32 bits; assim, é bem provável que, se você usar Windows, por exemplo, deva baixar o instalador para “Windows x86-64”. Se for um usuário de um Apple, tome a opção ao lado esquerdo desta.

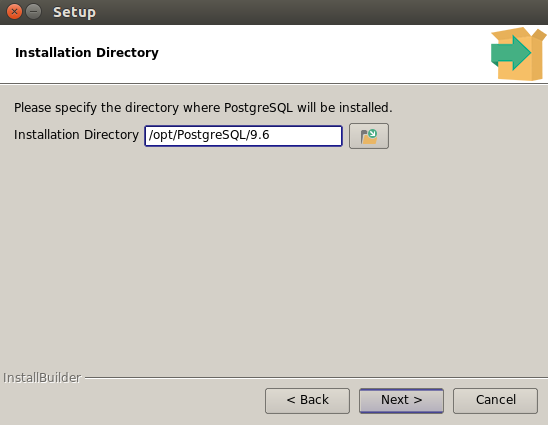


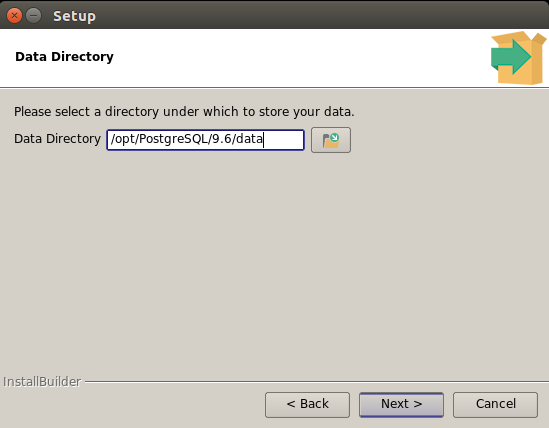
Algumas etapas do processo. Pode seguir as configurações-padrão e avançar. Será necessário definir uma senha no meio do processo - guarde ela!

Next...

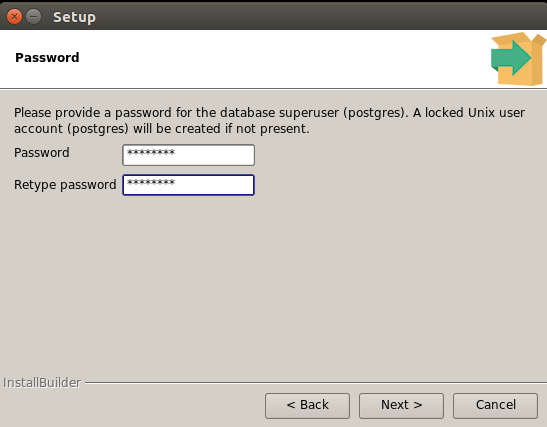
**

Se estiver usando o Windows, seus diretórios de instalação e de dados podem estar diferentes; pode deixar como estão por padrão. Dois Next seguidos.

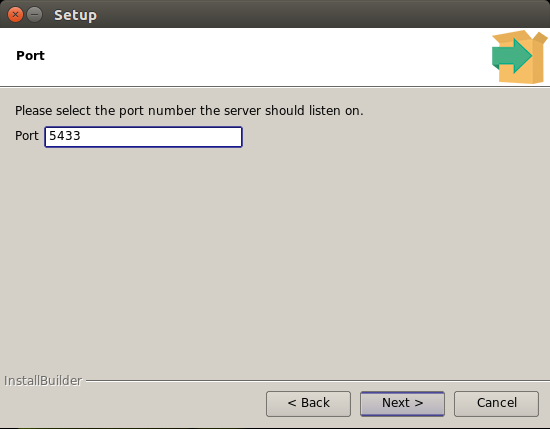




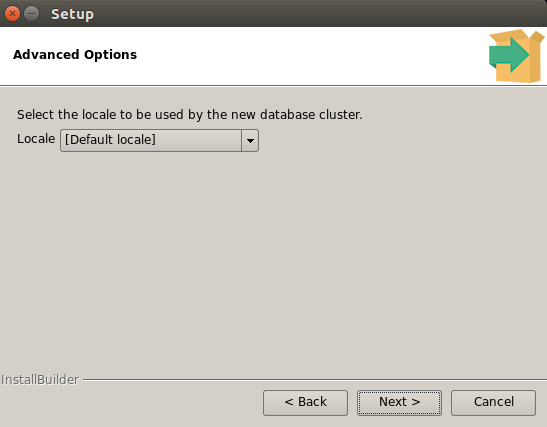
Agora você precisará definir uma senha para seu banco de dados. Ela será usada toda vez que for se conectar a ele. Guarde bem!

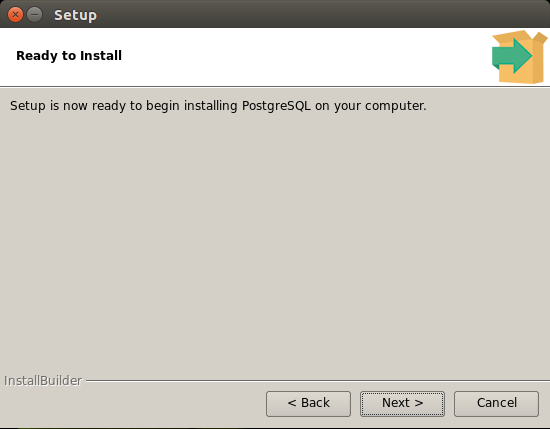


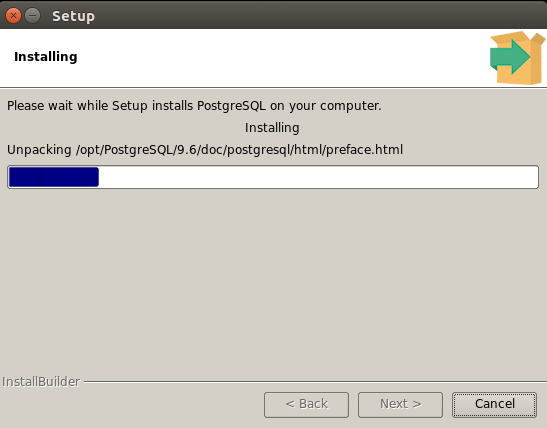
Pode deixar a porta-padrão de acesso. Next.



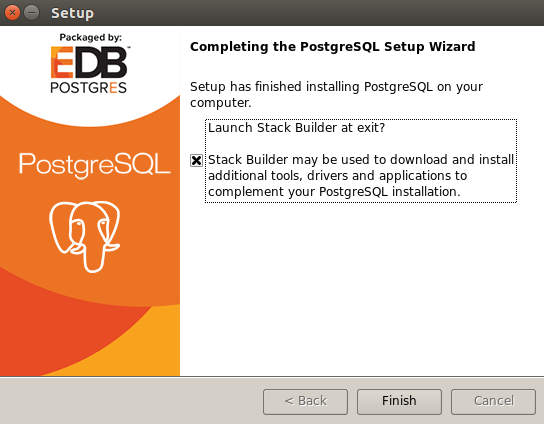
Três Next seguidos...







Ao cabo, pode desmarcar a opção de instalação do Stack Builder por ora. Finish.



Pronto. Após essa sequência com o instalador do Postgres - que, aliás, é a basicamente a mesma no Windows, no Linux e no Mac -, já temos nosso banco instalado. Por padrão, toda vez que você reiniciar o computador, ele será iniciado automaticamente, o que nos poupará trabalho no futuro!

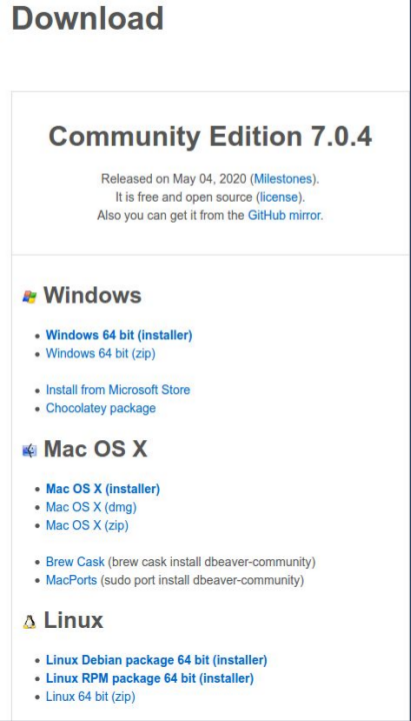
Ok, agora vamos instalar nosso cliente, o qual nos permitirá, de forma fácil, acessar e modificar este banco (e praticamente qualquer outro, local ou não).

## Instalando o cliente

Vamos instalar o DBeaver (open source e gratuito) - mas há várias outras opções no mercado. Siga o link abaixo e baixe a versão Community 7.0.4 para o seu sistema operacional.

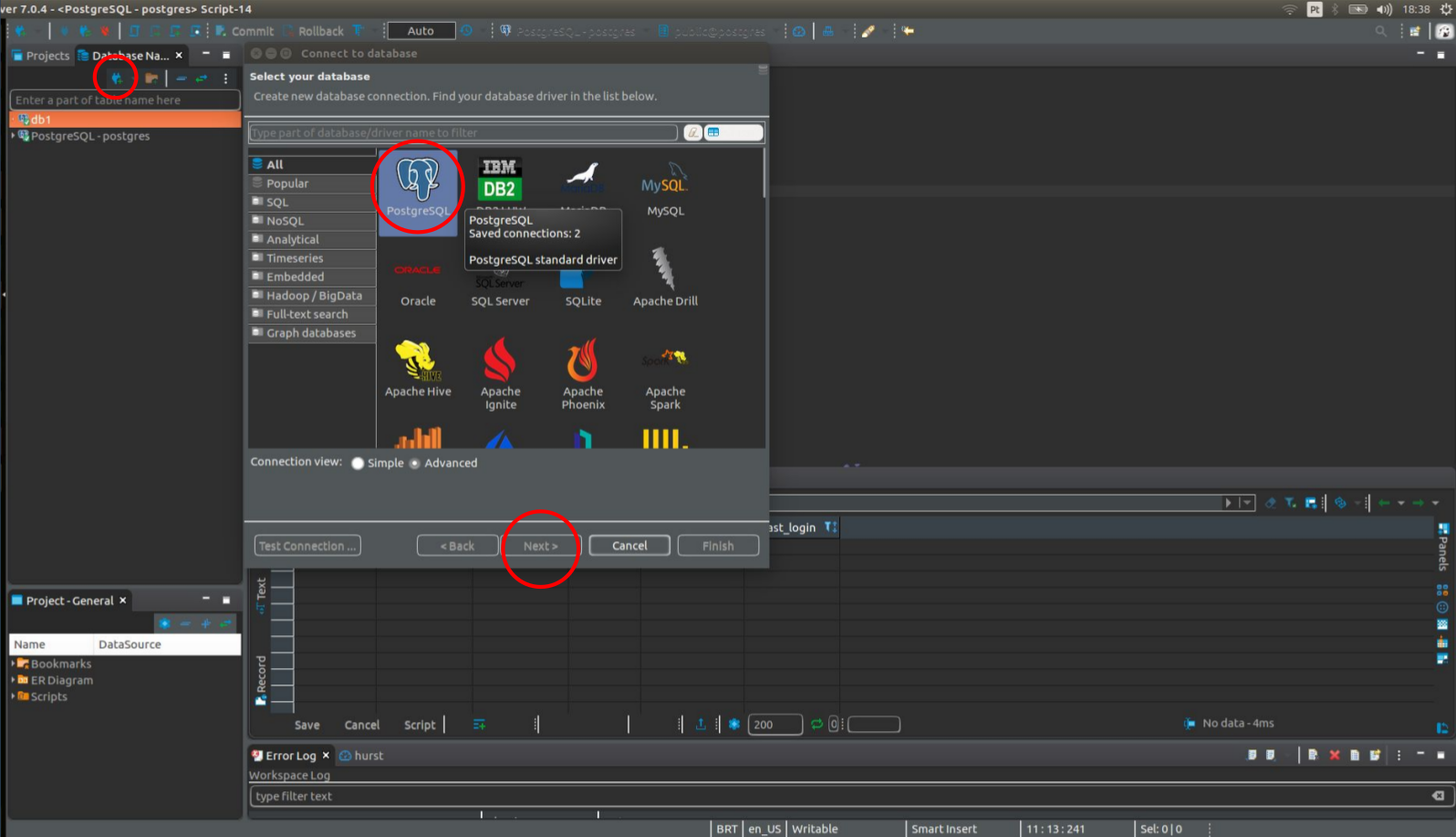
<https://dbeaver.io/download/>

O processo é simples. Pode deixar as configurações-padrão selecionadas.



## Conectando o cliente no servidor local

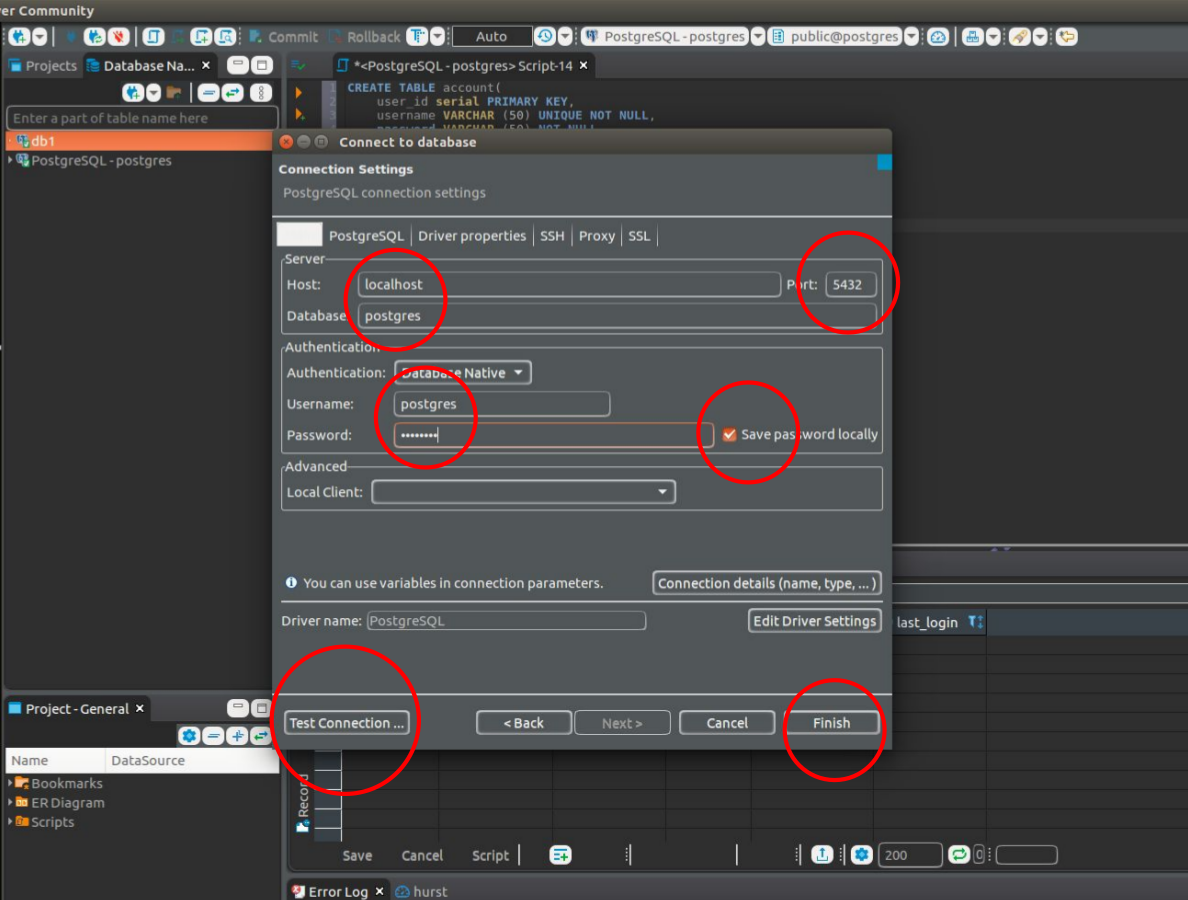
Após baixar o DBeaver, abra-o e clique para adicionar uma nova conexão, como demarcado pelo primeiro círculo vermelho superior na imagem abaixo. Selecione opção "PostgreSQL" e avance.



Na tela seguinte, nos é pedido para preencher os parâmetros de conexão, o "endereço do banco". Pode manter o host como "localhost" e a porta como "5432".

O database padrão do postgres se chama "postgres"; o username também. Mantenha-os assim, portanto.

Preencha a senha que você configurou para seu banco e clique em Test Connection. Convém deixar sua senha salva.



Se o teste de conexão for bem-sucedido, basta clicar em Finish. Do contrário, cheque as informações colocadas.

Pronto! Já temos um banco local e uma interface gráfica para modificá-lo e consultá-lo!

# Consultando tabelas com SQL

Vamos começar fazendo uma analogia com uma planilha de cálculo, como o Excel ou o Google Sheets.

Como no Excel, um banco relacional possui tabelas, que seriam as planilhas da nossa planilha de cálculo. Nessa comparação, um banco com quatro tabelas, Customers, Orders, Products e OrderDetails, seria como um arquivo do Excel com 4 planilhas, como abaixo.

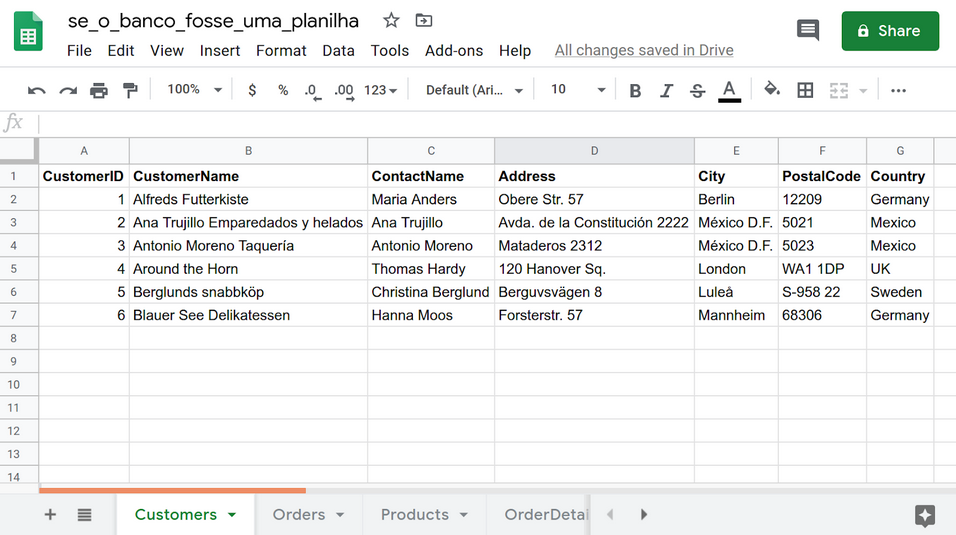
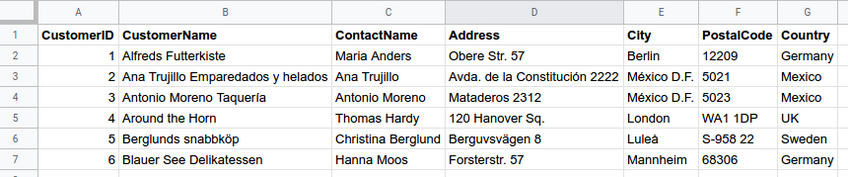


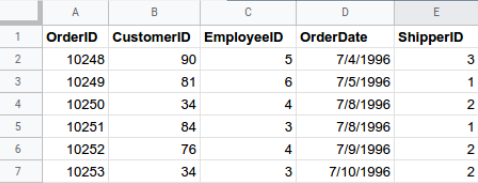
Tabela de clientes:



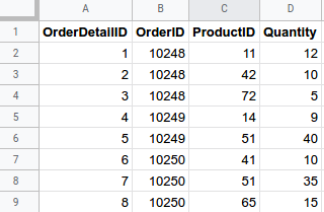
...de produtos:



...de pedidos:



...e de detalhes de cada pedido:



Por que não usar o Excel como “banco” então?!

* Escalabilidade
* Segurança
* Tipagem das colunas
* Ausência de uma linguagem expressiva o suficiente para simplificar consultas complexas
* Mistura entre software e banco: a planilha é o próprio software
* ACID:
  + Atomicity: cada transação ou ocorre ou falha completamente
  + Consistency: operações das transações têm de ser válidas
  + Isolation: transações concorrentes são sequenciais
  + Durability: transações efetuadas são não-voláteis

Ademais, tente responder às seguintes perguntas com o Excel. Embora certamente seja possível, dá um baita trabalho, principalmente porque precisamos ligar inúmeras tabelas de forma consistente, agrupas dados, etc.

* Quantos produtos em média um cliente compra da empresa ao longo de todo o seu lifetime?
* Para quais países mais se vendeu no primeiro trimestre de 1996?
* Como se comparam o ticket médio dos clientes alemães com aquele dos clientes mexicanos?
* Quantos produtos são vendidos em média nos meses de janeiro pelos 10 melhores funcionários?
* Ordene, nesta sequência, os produtos mais vendidos por país, cidade, funcionário.

## Tipos

Todas as colunas de cada tabela de um banco relacional tem, de partida, uma tipagem definida. Tipos variam um pouco de banco relacional para banco, mas, via de regra, há os seguintes:

* Booleanos: True/False
* Para texto: Char(10), Varchar(10), Text
* Numéricos: Inteiros e Floats
* Temporais: Date, Time, Timestamp
* E outros: JSON, Arrays, UUIDs, etc.

## Sintaxe básica

Vamos começar descrevendo como consultar uma tabela em SQL. Nas seções posteriores, expandiremos mostrando como consultar múltiplas tabelas.

### SELECT ... FROM ...

Considere que a tabela que iremos consultar se chama “Customer”, como no exemplo do Excel. Para consultar a tabela inteira, com todas as linhas e colunas, fazemos:

SELECT \*

FROM Customers;

Notas:

* Letra maiúscula para facilitar a leitura
* “;” no fim

### LIMIT / TOP N

Para limitar o número de linhas extraídas para não onerar o banco, temos (para as primeiras 10 linhas):

SELECT \*

FROM Customers

LIMIT 10;

Outra variante:

SELECT TOP 10 \*

FROM Customers;

### Filtro de colunas

Não precisamos puxar todas as colunas, se só precisamos de 2 ou 3:

SELECT CustomerName,

Address,

Country

FROM Customers;

### Alias: AS

Podemos renomear as colunas que serão exibidas na saída:

SELECT Country AS pais,

CustomerName AS cliente

FROM Customers;

### ORDER BY

Vamos ordenar a nossa lista de clientes pelo nome do país, inverso à ordem alfabética:

SELECT CustomerName,

Address,

Country

FROM Customers

ORDER BY Country DESC;

Notas:

* Para deixar em ordem alfabética, usar ASC em vez de DESC ou deixar vazio.

### DISTINCT

Quais são todos os países distintos na tabela?

SELECT DISTINCT Country

FROM Customers;

Quais são todas as duplas de cidades e países distintos na tabela?

SELECT DISTINCT City, Country

FROM Customers;

### WHERE

Liste todos os clientes da Alemanha:

SELECT \*

FROM Customers

WHERE Country = “Germany”;

### AND, OR, IN, LIKE

Liste todos os clientes da Alemanha que moram em Berlin ou Leipzig:

SELECT \*

FROM Customers

WHERE Country = “Germany” AND

City = “Berlin” OR City = “Leipzig”;

Ou,

SELECT \*

FROM Customers

WHERE Country = “Germany” AND

City IN (“Berlin”, “Leipzig”);

Podemos também listar todos os clientes de países cujos nomes começam com "M":

SELECT \*

FROM Customers

WHERE Country LIKE “M%”;

Nas próximas seções veremos como agregar, contar, somar, etc. em uma tabela.

# Construindo relatórios

As cláusulas que vimos anteriormente são os tijolos elementares para realizarmos quaisquer consultas em tabelas num banco relacional. Embora individualmente elas sejam bastante simples, sua combinação é justamente o que torna o SQL tão expressivo e poderoso (e - muitos concordariam - amado).

Todavia, é fácil perceber que não conseguimos, com o que temos até agora, construir relatórios em quaisquer situações. Se o que temos em mãos é uma tabela com, digamos, a quantidade de pessoas infectadas, recuperadas, etc., por país, bem, podemos usar o que foi aprendido para trabalhar sobre este relatório, filtrá-lo... Mas e se tudo o que temos no começo for uma tabela, linha após linha, com informações de cada paciente? Por exemplo:

idpaciente | idade | ... | \*\*país\*\* | infectado | recuperado | ...

Diferente do primeiro caso, no qual tivéssemos 100 países e 2 milhões de pacientes envolvidos, teríamos ainda 100 linhas, agora, nesta última tabela, teríamos 2 milhões de linhas na tabela. Nela, o menor "grão" é o paciente, não o país, o estado, ou a cidade... enfim, neste caso, não temos mais uma entidade agregadora.

Como relatórios e gráficos são agregados de informações, precisamos aprender a juntar os dados de tabelas de "fatos", como a última descrita, em tabelas que sumarizem esses dados, contando, somando, tomando a média, etc. É isso o que vamos aprender aqui.

### MIN(), MAX()

Começamos mostrando como obter os menores e os maiores valores de determinada coluna, de modo geral, ou em relação a outra coluna.

Para obter os valores mínimos de uma coluna, usamos:

SELECT MIN(Price)

FROM Products;

Podemos também listar os produtos mais caros por categoria

SELECT CategoryID,

MAX(Price)

FROM Products

GROUP BY 1;

Perceba, no último caso, a presença do termo GROUP BY 1 (o valor 1 se refere ao primeiro item que aparece no SELECT, mas poderíamos ter escrito também GROUP BY CategoryID). Ele instrui o banco de dados a agregar sobre a coluna CategoryID, buscando o maior preço, relativo a ela. Omitir esse termo gerará um erro na saída e, simultaneamente, é exigido, sob a mesma forma, com todas as funções agregadoras, como veremos adiante.

### COUNT()

Podemos contar todas as linhas de uma tabela de um modo bem simples:

SELECT COUNT(\*)

FROM Customers;

Já para contar o número de registros por país e ordenar pela maior contagem:

SELECT PostalCode AS CodPostal,

COUNT(\*) AS Qtd

FROM Customers

GROUP BY 1

ORDER BY 2 DESC;

### SUM()

Para somar todos os valores por categoria, usamos o SUM(). Abaixo, somamos os preços de todos os produtos da tabela por categoria.

SELECT CategoryID,

SUM(Price)

FROM Products

GROUP BY 1;

### AVG()

Com frequência também precisamos tomar a média para cada conjunto de categorias na nossa base. Para tanto, temos o AVG(). A seguir, calculamos a média de preços de produtos para cada categoria:

SELECT CategoryID,

AVG(Price)

FROM Products

GROUP BY 1;

### HAVING

E se quisermos filtrar os resultados de uma query pelas contagens ou somas calculados nessa própria query? O WHERE não vai funcionar, porque ele não é capaz de filtrar resultados de funções agregadoras, como COUNT, MIN ou SUM. Para isto, existe o HAVING.

Abaixo ilustramos seu uso como filtro: após contarmos quantos itens há em cada categoria, exibiremos apenas aquelas linhas de categorias que tenham, associadas a si, mais do que 10 itens ao menos:

SELECT CategoryID,

COUNT(\*)

FROM Products

GROUP BY 1

HAVING COUNT(\*) > 10;

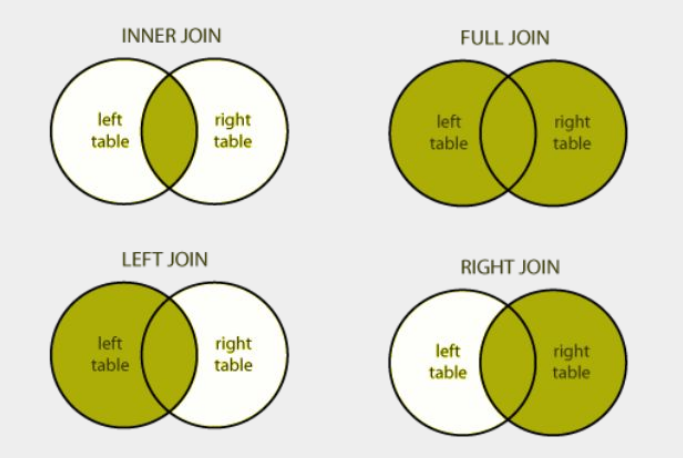
# Combinando resultados de múltiplas tabelas

Até agora vimos as aplicações da sintaxe do SQL em consultas de dados em uma única tabela. Embora seja já bastante útil, o cenário real é bem mais complexo. Na prática, uma empresa pode ter dezenas, centenas de tabelas em seu banco de dados.

Na verdade, com mais frequência ainda hoje em dia com o paradigma de microsserviços em alta, uma empresa em geral possui dezenas de bancos de dados em sua arquitetura e não há qualquer obrigação de esses bancos serem todos Postgres ou sequer relacionais. O cenário de múltiplos bancos heterogêneos é bastante comum.

Aqui vamos mostrar como fazemos para conectar dados que estão espalhados por mais de uma tabela num banco de dados relacional. Vamos começar pelos mais famosos representantes desta categoria: JOINs!

## JOINs



O diagramas acima ilustram, usando a linguagem da Teoria dos Conjuntos, a lógica por trás de cada um dos principais tipos de JOINs. Vamos ver cada um deles mais de perto.

### INNER JOIN

Com o INNER JOIN, conectamos duas tabelas pelo que há de comum em ambas (através da chaves usadas, claro; neste caso, as CustomerID, colunas presentes em ambas as tabelas). Abaixo, ligamos as tabelas Orders e Customers via este tipo de join.

SELECT Orders.OrderID,

Customers.CustomerName

FROM Orders

INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;

Ou, o que é uma boa prática, usando aliases para as tabelas:

SELECT o.OrderID,

c.CustomerName

FROM Orders AS o

INNER JOIN Customers AS c ON o.CustomerID = c.CustomerID;

### LEFT JOIN

Com o LEFT JOIN, obtemos tudo que está na primeira tabela e, deste conjunto, o que der match na segunda tabela. A sintaxe é praticamente idêntica à usado no caso do join anterior:

SELECT o.OrderID,

c.CustomerName

FROM Orders AS o

LEFT JOIN Customers AS c ON o.CustomerID = c.CustomerID;

### RIGHT JOIN

Neste caso, obtemos tudo que está na segunda tabela e, deste conjunto, o que der match na primeira tabela - o contrário do LEFT JOIN (por isto, em geral, não há por que usar o RIGHT JOIN se você pode ir com o LEFT JOIN - de fato é isto que ocorre na prática).

SELECT o.OrderID,

c.CustomerName

FROM Orders AS o

RIGHT JOIN Customers AS c ON o.CustomerID = c.CustomerID;

### FULL JOIN

Também descrito como FULL OUTER JOIN. Com ele consultamos tudo o que está em ambas as tabelas - ou, na linguagem da teoria de conjuntos, a união do seus conteúdos.

SELECT o.OrderID,

c.CustomerName

FROM Orders AS o

FULL JOIN Customers AS c ON o.CustomerID = c.CustomerID;

Desses todos, os dois primeiros, os INNER JOIN e LEFT JOIN, são provavelmente os mais usados na prática.

Além disso, embora as operações acima estejam descritas para duas tabelas, elas foram concebidas para conectar um número arbitrário delas, mas sempre aos pares, repetindo a estrutura.

## UNIONs

As operações de UNION no SQL servem para concatenar verticalmente os conteúdos de duas tabelas.

### UNION

A operação de UNION, mantém apenas valores distintos das colunas exibidas em ambas as tabelas. Sua sintaxe é:

SELECT City FROM Customers

UNION

SELECT City FROM Suppliers

A query acima constrói uma tabela (que, neste caso, é de apenas uma coluna, City), na qual estão listadas todas as cidades que aparecem em ambas as colunas City das duas tabelas, Customers e Suppliers, sem repetir nomes de cidades.

### UNION ALL

A UNION ALL é bastante semelhante à operação anterior, com a diferença de que não se preocupa em ter elementos únicos (no exemplo abaixo, cidades únicas). Ela simplesmente "empilha" os registros das duas tabelas.

SELECT City FROM Customers

UNION

SELECT City FROM Suppliers

# Construindo um banco de dados

Nesta seção vamos falar sobre os elementos fundamentais de um banco de dados relacional; a saber, em ordem hierárquica: databases, schemas e tables. Destes, as últimas são o objeto central: é nas tabelas que persistimos nossos registros, em linhas e colunas.

## Database

Por mais que possa parecer estranho um único banco de dados físico pode conter vários “bancos de dados” (databases) virtuais independentes. Para criar um database, fazemos:

CREATE DATABASE letscode;

Para renomeá-los, temos:

ALTER DATABASE letscode RENAME TO lets\_code;

Para deletá-lo,

DROP DATABASE [IF EXISTS] lets\_code;

## Schemas

Schemas são conjuntos de tabelas num banco de dados e servem para nos ajudar a organizá-las logicamente.

Para criar schemas, fazemos:

CREATE SCHEMA letscode;

Para renomeá-los, temos:

ALTER SCHEMA letscode RENAME TO lets\_code;

Para deletá-lo,

DROP SCHEMA [IF EXISTS] lets\_code CASCADE;

## Tables

Tabelas são o elemento central de um banco relacional. Para criar uma tabela "customers", para guardar os dados dos nossos clientes, fazemos:

CREATE TABLE customers (

customerid INTEGER PRIMARY KEY,

customername varchar(255),

contactname varchar(255),

address TEXT,

email VARCHAR (50) UNIQUE,

city varchar(255),

postalcode varchar(255),

country varchar(255) NOT NULL

);

Repare na estrutura exibida no código acima. Cada linha corresponde a uma coluna a ser criada na tabela e, ao lado direito do nome de cada coluna, declaramos o tipo que esta coluna suportará. Por exemplo, na linha

customername varchar(255),

estamos criando uma coluna chamada customername para receber strings de até 255 caracteres. Já no caso da última linha:

country varchar(255) NOT NULL

além de definirmos o tipo da coluna como strings, também estamos proibindo, de partida, que qualquer valor a ser inserido nessa coluna possa ser nulo (NULL).

Outra situação recorrente acontece na linha:

email VARCHAR (50) UNIQUE,

Na qual proibimos o banco de salvar emails repetidos de clientes.

### Sobre chaves

Se prestou atenção no CREATE TABLE anterior, percebeu que não falamos daquilo que surge logo na primeira coluna:

customerid INTEGER PRIMARY KEY,

Esta instrução diz ao banco de dados para criar uma coluna chamada customerid que será a chave primária desta tabela. Esta chave é um identificador único para cada registro na tabela - portanto, não pode se repetir e muito menos ser nula. Grosso modo, servem às linhas da tabela como as placas de identificação servem aos carros.

Em geral, as chaves primárias são números inteiros sucessivos; no entanto, em determinadas aplicações em que é difícil garantir a unicidade das chaves, como em aplicações offline first que precisam garantir consistência mesmo em casos de intermitência de conexão com a internet, também podem ser usados [UUIDs](https://en.wikipedia.org/wiki/Universally_unique_identifier) para esta tarefa.

Embora não seja obrigatório configurar uma chave primária quando se projeta uma tabela de banco, é uma boa prática fazê-lo.

Quando precisarmos criar referências claras, vínculos, entre colunas de uma tabela e colunas de outra, entram em cena as chaves estrangeiras. Suponha que estejamos criando uma tabela de pedidos, orders, como mostramos abaixo:

CREATE TABLE orders (

orderid SERIAL PRIMARY KEY,

customerid INTEGER REFERENCES customers(customerid),

employeeid INTEGER REFERENCES employees(employeeid),

orderdate DATE

);

Repare que, no cenário esboçado acima, um pedido está sempre relacionado a um cliente que o fez e a um funcionário que o vendeu. Portanto, podemos criar vínculos entre cada ID de pedido (orderid) com os respectivos IDs de cliente (customerid) e de funcionário (employeeid), onde o primeiro "vive" (é uma coluna) na tabela customers e o outro vive na tabela employees - ou seja, não é possível sequer criar a tabela acima se, antes, não tivermos já criado essas duas, cada qual com esses IDs.

Por conta desse vínculo criado entre as três tabelas, não será possível deletar um cliente da tabela customers se ele tiver associado a ele pedidos na tabela orders. Antes é preciso apagar todos os seus rastros vinculados no banco. Conquanto isto pode ser tedioso para o analista de dados, é também uma forma explícita de garantir que inconsistências não sejam criadas entre as informações conectadas em múltiplas tabelas no banco.

Se, no entanto, interessa-lhe mais dar instruções gerais e garantir que o banco "resolva as pendências, não importam as consequências", pode usar, na criação da tabela, o recurso do ON DELETE CASCADE:

CREATE TABLE orders (

orderid SERIAL PRIMARY KEY,

customerid INTEGER REFERENCES customers(customerid) ON DELETE CASCADE,

employeeid INTEGER REFERENCES employees(employeeid) ON DELETE CASCADE,

orderdate DATE

);

Neste caso, por exemplo, ao deletar um cliente ou funcionário nas suas respectivas tabelas, todas as transações das quais ele tomou parte na tabela orders serão deletadas juntamente.

...Um adendo: percebeu que usamos um "tipo" diferente para declarar a coluna orderid nesta tabela?

orderid SERIAL PRIMARY KEY,

Este SERIAL basicamente automatiza o processo de criação incremental de chaves primárias para a tabela. Assim, ao inserir novos registros, não é necessário inserir os IDs dos pedidos.

### Alterando tabelas existentes

Para renomear uma tabela existente, temos:

ALTER TABLE customers RENAME TO new\_customers;

E para deletar a tabela:

DROP TABLE IF EXISTS new\_customers CASCADE;

Para deletar de forma eficiente todo o conteúdo de uma tabela, podemos fazer:

TRUNCATE customers;

Ou, para resetar também os índices, temos:

TRUNCATE TABLE customers RESTART IDENTITY;

Podemos mudar o tipo das colunas também!

ALTER TABLE customers

ALTER COLUMN address TYPE TEXT;

Ou adicionar novas colunas:

ALTER TABLE customers

ADD COLUMN hired BOOLEAN;

Inversamente, para eliminar um coluna, temos:

ALTER TABLE customers

DROP COLUMN hired;

# Manipulando dados no banco

Instalamos o banco, o cliente e construímos a estrutura básica para receber os dados. Se fôssemos construir uma casa, agora estaríamos prestes a preenchê-la com móveis e tudo o que a torna única. Pois vamos então inserir, atualizar e deletar dados das nossas tabelas!

## INSERT

Para inserir novos registros numa tabela, temos o INSERT:

INSERT INTO customers(customerid, customername, contactname, address, city,

postalcode, country)

VALUES (1, 'Alfreds Futterkiste', 'Maria Anders', 'Obere Str. 57', 'Berlin',

'12209', 'Germany');

Como criamos a tabela acima com uma chave primária incremental, podemos omitir a coluna customerid da lista:

INSERT INTO customers(customername, contactname, address, city, postalcode,

country)

VALUES ('Alfreds Futterkiste', 'Maria Anders', 'Obere Str. 57', 'Berlin',

'12209', 'Germany');

Experimente dar um INSERT sem um dos campos (além da chave), e note o que ocorre na tabela:

INSERT INTO customers(customername, contactname, city, postalcode, country)

VALUES ('Alfreds Futterkiste', 'Maria Anders', 'Berlin', '12209', 'Germany');

Segue:

SELECT \*

FROM customers;

## UPDATE

Se quisermos atualizar um ou vários registros, usamos o UPDATE:

UPDATE customers

SET city = 'Sao Paulo'

WHERE customerid = 3;

É importante controlar o tamanho da atualização com o WHERE! A seguinte query atualiza a coluna “city” de todos os registros na tabela:

UPDATE customers

SET city = 'Sao Paulo';

## DELETE

Finalmente, para deletar um ou vários registros, temos:

DELETE FROM customers

WHERE address IS NULL;

Assim como no caso do UPDATE, é extremamente importante dizer quem será deletado através do WHERE. Na ausência deste filtro, todo o conteúdo \*da tabela será \*apagado!

DELETE FROM customers;

## COPY

O Postgres possui uma operação especial chamada COPY que nos permite inserir e retirar dados em batches, isto é, carregar ou baixar, por exemplo, CSVs inteiros no/do nosso banco de dados.

### COPY ... FROM

Para carregar um CSV arquivo.csv inteiro na tabela, digamos, customers, temos:

COPY customers

FROM 'C:\endereco\arquivo.csv' WITH

DELIMITER ','

CSV

HEADER;

Acima, DELIMITER informa ao banco qual o delimitador usado no CSV (em geral, ",", ";" ou pipe, "|"); CSV diz que informa o formato do arquivo de entrada e HEADER diz ao banco para ignorar a primeira linha do arquivo, o header, porque ele basicamente contém os nomes das colunas.

### COPY ... TO

Uma variação da operação anterior também nos permite tirar os dados do banco, salvando-os como CSV (no caso abaixo, no diretório C:\Users\Public):

COPY (SELECT \* FROM orders)

TO 'C:\Users\Public'

DELIMITER ','

CSV

HEADER;

Repare na primeira linha que a operação é versátil o suficiente para nos permitir baixar o resultado não só de uma tabela do banco, mas de qualquer query que queiramos.

# Modelando problemas reais em bancos relacionais

Até agora tomamos uma atitude mais pragmática do ponto de vista de um analista de dados: assumimos que, qualquer que seja nosso banco de dados, ele já estava lá quando chegamos: a nós cabia apenas escrever as consultas para extrair informações, ou inserir ou editar aquelas que achávamos pertinentes. É como se alugássemos uma casa: não nos preocupamos em construí-la, mas apenas em mobiliar e morar.

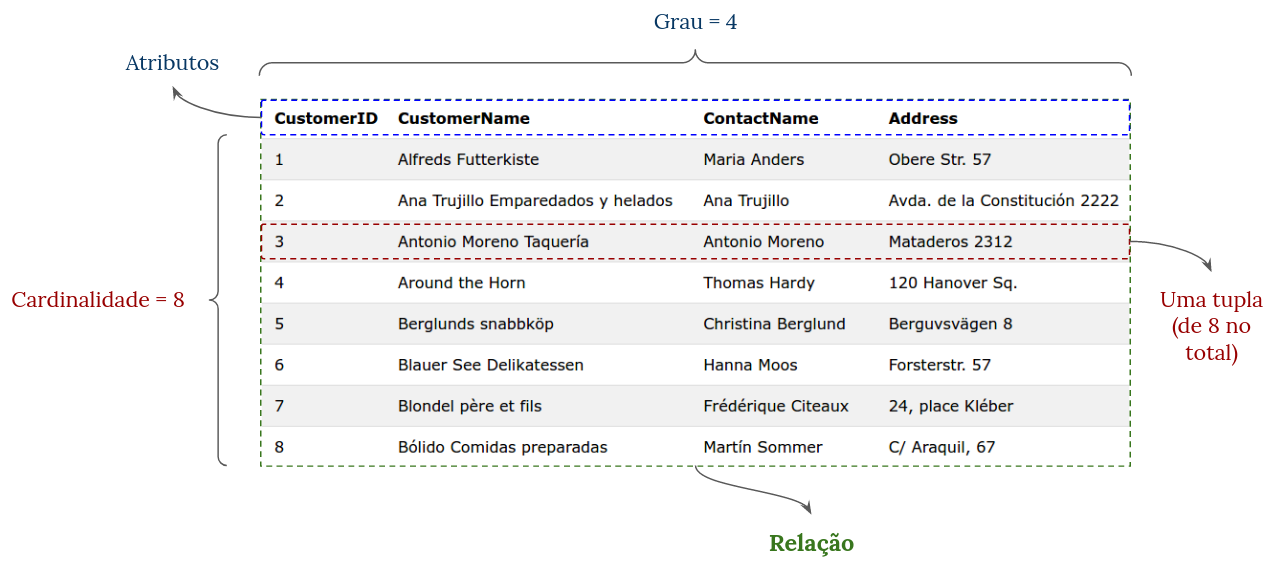
Nesta seção daremos um passo para trás, e nos dedicaremos a pensar a "arquitetura" dessa casa: como definimos as tabelas e seus vínculos a partir do problema de negócio que desejamos modelar? Para isto, vamos falar um pouco do conceito de modelo relacional e mostraremos como aplicá-lo na prática para construir um banco do zero.

## Um pouco da terminologia

O modelo relacional foi primeiramente formulado em 1969 pelo cientista inglês Edgar Codd. Formalmente, ele é definido em termos de lógica de primeira ordem e teoria dos conjuntos, e possui uma construção bem elaborada por trás, o que garante sua consistênca. Aqui, no entanto, vamos nos focar na sua face prática.

Primeiro, seu nome advém do conceito de "relação" da lógica. Por incrível que pareça, ele não se refere às relações expressas nas chaves estrangeiras das tabelas; em vez disso, podemos entender uma "relação" como a própria tabela (há algum abuso de linguagem aqui, mas vamos manter assim por simplicidade). Em inglês é um pouco mais fácil distinguir, porque "relationship" se refere às relações entre as entidades modeladas, enquanto "relation" se refere às próprias tabelas no modelo relacional.

No modelo relacional, as colunas da tabela (= relação) são chamadas de atributos; as linhas, são tuplas. Dizemos que o número de atributos da relação é o seu grau; já o seu total de linhas, a sua cardinalidade. A figura abaixo resume a nomenclatura.



Uma relação é composta de duas partes: um schema e uma instância:

* Schema da relação: define o nome da relação e de seus atributos, além dos tipos destes últimos; ex.:

funcionarios(id INTEGER, nome completo VARCHAR, endereco VARCHAR, idade INTEGER, genero CHAR, salario NUMERIC)

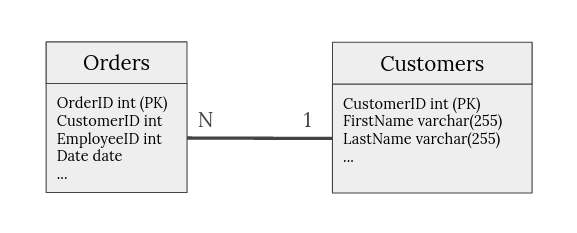
* Instância da relação: a tabela com seus atributos e tuplas.

## Relacionamentos entre tabelas

Na seção de consultas em múltiplas tabelas, você deve ter percebido que há diferentes tipos de relacionamentos entre tabelas num banco relacional. Vejamos cada um:

### One-to-many (1-N)

Numa tabela de pedidos no banco de um ecommerce, por exemplo, espera-se que o ID de um cliente possa aparecer diversas vezes, porque, de fato, um mesmo cliente pode fazer vários pedidos. A figura abaixo ilustra esse relacionamento entre as duas tabelas. Dizemos que se trata de um relacionamento one-to-many ou um para muitos, e podemos escrever N para 1, como no diagrama.



Veremos que este é o relacionamento mais importante.

### One-to-one (1-1)

Neste tipo, cada registro em uma tabela se relaciona com apenas um registro na outra, e vice-versa. Por exemplo, uma tabela de países e outra de capitais: cada país só tem uma capital, e cada capital obrigatoriamente só o é se for de um país.

Relacionamentos 1-1 podem ser convertidos em uma única tabela; no caso anterior, em vez de duas tabelas, poderíamos ter apenas uma tabela "País" com uma coluna chamada "capital". No entanto, às vezes preferimos manter conjuntos de dados de uma tabela separados (talvez porque sejam duas tabelas imensas e não precisamos de parte das colunas sendo "arrastada" em todas as queries). Neste caso, o relacionamento 1-1 deixa de ser supérfluo.

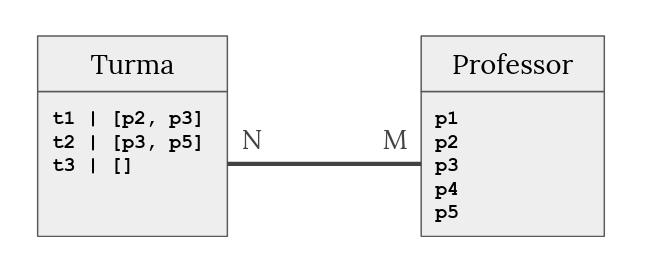
### Many-to-many (N-M)

Este relacionamento serve para modelar situações nas quais a entidade representada numa tabela pode estar associada a múltiplas entidades expressas na segunda tabela. Por exemplo, a relação entre escritores e livros: um escritor pode escrever vários livros, mas é admissível também que livros tenham mais de um autor.

Embora haja muitas relações N-M no mundo real, o modo como implementamos elas num banco relacional é pela sua decomposição em 2 relacionamentos do tipo 1-N com uma terceira tabela.

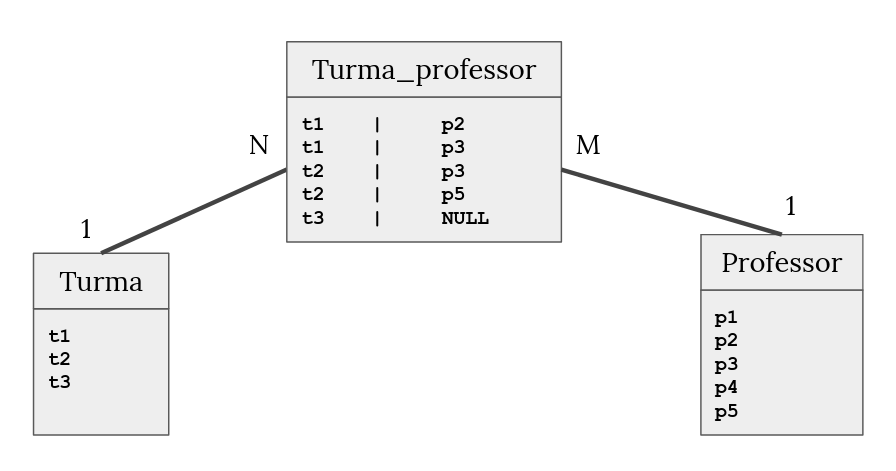
Tomemos o exemplo do relacionamento entre uma tabela de turmas e uma de professores em uma escola. De novo, um professor pode dar aulas em mais de uma turma, e uma turma também pode ter aulas com mais de um professor. Note que agora não é possível mais apenas adicionar uma chave estrangeira ìdProfessor na tabela professor para resolver este problema.

Poderíamos ainda construir o modelo abaixo, no qual, para cada turma t1, t2, ..., admite-se um vetor de professores associado [p1, p2, ...]. No que à t1 estão associados p2 e p3 e, inversamente, p3 dá aulas em t1 e t2.



Esta solução, porém, é ruim. Além de ferir princípios de normalização de tabelas, como faríamos um JOIN entre as duas tabelas?

Assim, para evitar este problema, criamos uma terceira tabela "artificial" para servir de junção das duas outras tabelas originais, como demonstra a próxima figura. Note que agora eliminamos os vetores em cada linha e decompusemos a relação many-to-many inicial em 2 relações one-to-many.



Portanto, para seguir com os JOINs entre as tabelas, precisamos agora passar antes pela tabela intermediária - a qual, vale ressaltar, também pode ter uma chave primária independente dos dois IDs que guarda.