PORQUE USAR POSTGRESQL

SE VOCÊ AINDA NÃO O FAZ.

@fnando

PORQUE USAR POSTGRESQL

SE VOCÊ AINDA NÃO O FAZ.

@fnando

NANDO VIEIRA


HEROKU

howto.

POR QUE NÃO USAR O MYSQL?

O MySQL não é o melhor banco de dados que você poderia estar usando. Entenda o porquê.

A COMUNIDADE SE DIVIDIU EM DUAS.

Com a aquisição da MySQL, surgiu um fork chamado MariaDB. Qual distribuição você irá usar?

O MYSQL É POBRE EM FUNCIONALIDADES.

Muitas funcionalidades implementadas por outros bancos de dados, incluindo SQLite, não existem no MySQL.

https://modern-sql.com/

SEM ROLLBACK EM DE MUDANÇAS NO DDL

Se você usar Ruby on Rails já deve ter enfrentado problemas de ter que desfazer migrações na mão.

```
BEGIN;

CREATE TABLE foo (
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY NOT NULL
);

ROLLBACK;

SHOW TABLES;

+----------------+
| Tables_in_test |
+----------------+
| foo            |
+----------------+
1 row in set (0.00 sec)
```

```
BEGIN;

CREATE TABLE foo (
  id serial NOT NULL PRIMARY KEY
);

ROLLBACK;
\d

      List of relations
 Schema | Name | Type | Owner
--------+------+------+-------
(0 rows)
```

INCONSISTÊNCIA DE DADOS

O MySQL é praticamente o JavaScript dos bancos de dados.

```
CREATE TABLE foo (
  text_col VARCHAR(1),
  numeric_col DECIMAL(1,1),
  date_col DATE
);

INSERT INTO foo (text_col, numeric_col, date_col)
         VALUES ("abc", 123.45, "LOL");

Query OK, 1 row affected, 3 warnings (0.00 sec)

SELECT * FROM foo;

+----------+-------------+------------+
| text_col | numeric_col | date_col   |
+----------+-------------+------------+
| a        |         0.9 | 0000-00-00 |
+----------+-------------+------------+
1 row in set (0.00 sec)
```

```
CREATE TABLE foo (
  text_col VARCHAR(1),
  numeric_col DECIMAL(1,1),
  date_col DATE
);

INSERT INTO foo (text_col) VALUES ('abc');
ERROR:  22001: value too long for type character varying(1)

INSERT INTO foo (numeric_col) VALUES (123.45);
ERROR:  22003: numeric field overflow
DETAIL:  A field with precision 1, scale 1 must round to an absolute value less than 1.

INSERT INTO foo (date_col) VALUES ('LOL');
ERROR:  22007: invalid input syntax for type date: "LOL"
```

```
SET global sql_mode = "TRADITIONAL";


INSERT INTO foo (text_col) VALUES ("abc");
ERROR 1406 (22001): Data too long for column 'text_col' at row 1

INSERT INTO foo (numeric_col) VALUES (123.45);
ERROR 1264 (22003): Out of range value for column 'numeric_col' at row 1

INSERT INTO foo (date_col) VALUES ("LOL");
ERROR 1292 (22007): Incorrect date value: 'LOL' for column 'date_col' at row 1

INSERT IGNORE INTO foo (text_col, numeric_col, date_col)
                VALUES ("abc", 123.45, "LOL");
Query OK, 1 row affected, 3 warnings (0.01 sec)
```

Dá para corrigir este comportamento... kinda. ¯_(ツ)_/¯

COMPARAÇÃO DE STRINGS PODE SER UM PROBLEMA

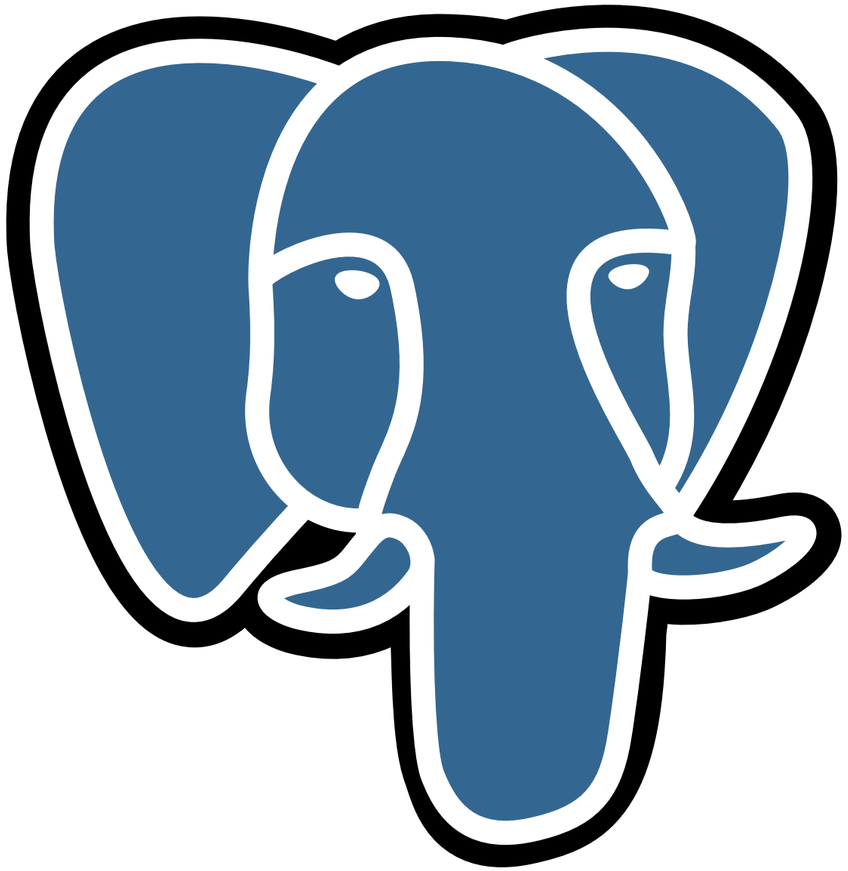
Isso pode levar à falhas de segurança dependendo de como o seu framework funciona.

```
SELECT * FROM users;

+----+------------------+
| id | email            |
+----+------------------+
|  1 | john@example.com |
|  2 | mary@example.com |
+----+------------------+
2 rows in set (0.00 sec)

SELECT * FROM users WHERE email = 0;

+----+------------------+
| id | email            |
+----+------------------+
|  1 | john@example.com |
|  2 | mary@example.com |
+----+------------------+
2 rows in set, 2 warnings (0.00 sec)
```

FULL-TEXT SEARCH. TRANSAÇÕES. ESCOLHA UM.

*Não, você não pode ter ambos.

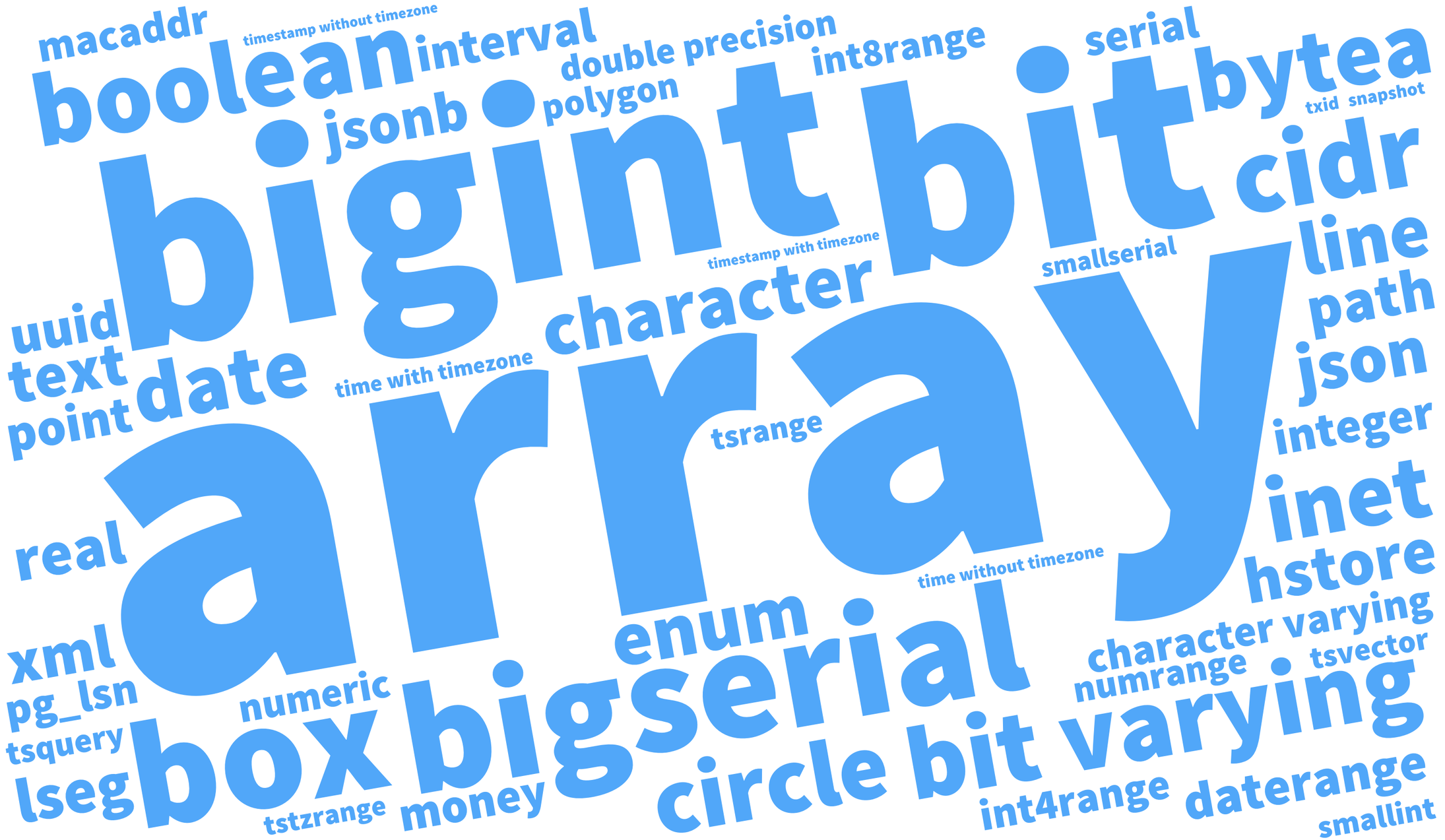
```
CREATE TABLE articles (
  content text not null
) ENGINE MyISAM;

BEGIN;
INSERT INTO articles (content) values ('Some text');

ROLLBACK;
Query OK, 0 rows affected, 1 warning (0.02 sec)


SHOW WARNINGS;

+---------+------+-----------------------------------------------------+
| Level   | Code | Message                                             |
+---------+------+-----------------------------------------------------+
| Warning | 1196 | Some non-transactional changed tables couldn't be rolled back |
+---------+------+-----------------------------------------------------+
1 row in set (0.01 sec)
```

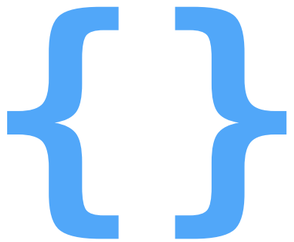


COM POSTGRESQL VOCÊ TEM TUDO ISSO E MUITO MAIS!

GRANDE DIVERSIDADE DE TIPOS DE CAMPOS

O PostgreSQL possui muitos campos nativamente, como array, json, uuid, e mais.


macaddr
timestamp without timezone
boolean
interval
double precision
int8range
serial
bytea
jsonb
polygon
txid
snapshot
bigint
bit
cidr
uuid
timestamp with timezone
smallserial
line
character
text
time with timezone
path
date
array
tsrange
json
point
integer
real
inet
xml
enum
time without timezone
hstore
character varying
pg_lsn
numeric
numrange
tsvector
box
bigserial
tsquery
circle
bit varying
lseg
tstzrange
money
int4range
daterange
smallint



SUPORTE PARA CAMPOS DO TIPO JSON

Você não precisa mais usar algo como *MongoDB para ser "schema-less".

*http://fnando.me/rb

JSON

PostgreSQL 9.2

Armazena o texto as-it-is

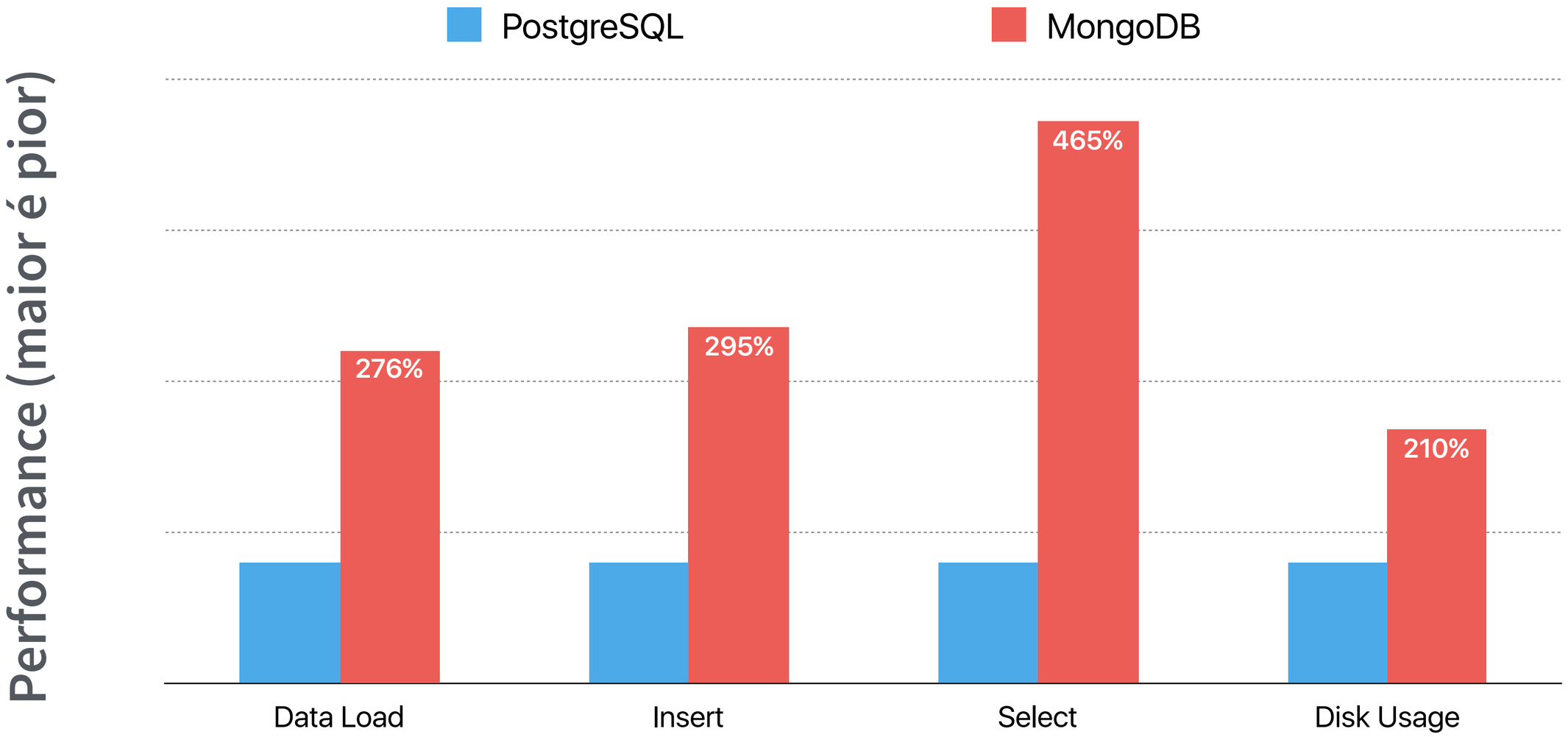
Não suporta índices

JSON-B

PostgreSQL 9.4

Armazena representação binária

Suporta índices



http://fnando.me/1e2

```ruby
class CreateUsers < ActiveRecord::Migration
  def change
    enable_extension "citext"

    create_table :users do |t|
      t.text :name, null: false
      t.citext :username, null: false
      t.jsonb :preferences, null: false, default: "{}"
    end

    add_index  :users, :preferences, using: :gin
  end
end
```

Suporte nativo no Rails à partir da versão 4.2

```ruby
user = User.create!({
  name: "John Doe",
  username: "johndoe",
  preferences: {
    twitter: "johndoe",
    github: "johndoe",
    blog: "http://example.com"
  }
})

# Show preferences.
user.preferences
#=> {"blog"=>"http://example.com", "github"=>"johndoe", "twitter"=>"johndoe"}

# Get blog.
user.preferences["blog"]
#=> http://example.com
```

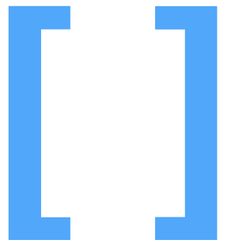
Usando o atributo JSON

```ruby
class User < ActiveRecord::Base
  store_accessor :preferences, :twitter, :github, :blog
end

user.twitter = "fnando"
```

Criando métodos de acesso às propriedades

```ruby
# preferences->newsletter = true
User.where("preferences @> ?", {newsletter: true}.to_json)

# preferences->interests = ["ruby", "javascript", "python"]
User.where("preferences -> 'interests' ? :language", language: "ruby")

CREATE INDEX preferences_interests_on_users
ON users
USING GIN ((preferences->'interests'))

# preferences->twitter AND preferences->github
User.where("preferences ?& array[:keys]", keys: ["twitter", "github"])

# preferences->twitter OR preferences->github
User.where("preferences ?| array[:keys]", keys: ["twitter", "github"])

# preferences->state = "SP" AND preferences->city = "São Paulo"
User.where("preferences @> ?", {city: "San Francisco", state: "CA"}.to_json)
```

Fazendo queries no campo jsonb



SUPORTE PARA CAMPOS DO TIPO ARRAY

Nem toda lista precisa estar em uma tabela separada de banco de dados.

```
SELECT * FROM articles;

 id |                           title                           |          tags
----+-----------------------------------------------------------+--------------------------------
  1 | Using ES6 with Asset Pipeline on Ruby on Rails            | {javascript,rails,ruby}
  2 | Creating generators and executables with Thor             | {ruby}
  3 | Creating custom Minitest reporters                        | {ruby,testing}
  4 | Using UUID with PostgreSQL and ActiveRecord               | {rails,postgresql,activerecord}
  5 | Using PostgreSQL and jsonb with Ruby on Rails             | {rails,postgresql}
  6 | Using PostgreSQL and hstore with Rails                    | {rails,postgresql}
  7 | Using insensitive-case columns in PostgreSQL with citext | {rails,postgresql}
  8 | Setting up Ember.js (ember-cli) with Rails                | {rails,javascript}
  9 | Setting up Ember.js with Rails                            | {rails,javascript}
(9 rows)
```

Selecione todos os posts

```
SELECT * FROM articles WHERE tags && ARRAY['ruby'];

 id |                      title                       |          tags
----+--------------------------------------------------+--------------------------
  1 | Using ES6 with Asset Pipeline on Ruby on Rails | {javascript,rails,ruby}
  2 | Creating generators and executables with Thor  | {ruby}
  3 | Creating custom Minitest reporters             | {ruby,testing}
(3 rows)
```

Selecione todos os posts marcados com ruby

```
SELECT * FROM articles WHERE tags && ARRAY['ruby', 'javascript'];
```

```
 id |                     title                      |          tags
----+------------------------------------------------+-------------------------
  1 | Using ES6 with Asset Pipeline on Ruby on Rails | {javascript,rails,ruby}
  2 | Creating generators and executables with Thor  | {ruby}
  3 | Creating custom Minitest reporters             | {ruby,testing}
  8 | Setting up Ember.js (ember-cli) with Rails     | {rails,javascript}
  9 | Setting up Ember.js with Rails                 | {rails,javascript}
(5 rows)
```

Selecione todos os posts marcados com ruby ou javascript

```
SELECT * FROM articles WHERE tags @> ARRAY['rails', 'postgresql'];
```

```
 id |                           title                            |              tags
----+------------------------------------------------------------+---------------------------------
  4 | Using UUID with PostgreSQL and ActiveRecord                | {rails,postgresql,activerecord}
  5 | Using PostgreSQL and jsonb with Ruby on Rails              | {rails,postgresql}
  6 | Using PostgreSQL and hstore with Rails                     | {rails,postgresql}
  7 | Using insensitive-case columns in PostgreSQL with citext | {rails,postgresql}
(4 rows)
```

Selecione todos os posts marcados com rails e postgresql

```sql
WITH tags AS (
  SELECT lower(unnest(tags)) AS tag
  FROM articles
)

SELECT
  tag,
  count(tag) AS occurrences
FROM tags
GROUP BY tag
```

```
     tag      | occurrences
--------------+-------------
 postgresql   |           4
 rails        |           7
 activerecord |           1
 javascript   |           3
 ruby         |           3
 testing      |           1
(6 rows)
```

Selecione as tags com contagem de ocorrências (tag cloud)

As colunas do tipo array só podem conter um tipo de dados, exclusivamente.
Não é possível misturar tipos de dados diferentes.

SUPORTE PARA UUIDS COMO CHAVE PRIMÁRIA

Evite todos os problemas de gerenciamento de chaves primárias sequenciais de um modo simples.

```
CREATE EXTENSION "uuid-ossp";


CREATE TABLE users (
  id uuid PRIMARY KEY NOT NULL DEFAULT uuid_generate_v4(),
  email citext NOT NULL UNIQUE
);

INSERT INTO users (email) VALUES ('john@example.com');


SELECT * FROM users;
                  id                  |      email
--------------------------------------+------------------
 a38a3f0a-ce50-405a-804b-2c85379f1c5a | john@example.com
(1 row)
```

Embora o Rails tenha suporte para uuid, ele não permite configurar facilmente o tipo de chaves primárias.

http://fnando.me/1e3

E VOCÊ AINDA PODE USAR TIPOS PERSONALIZADOS.

Com extensões, é possível adicionar tipos personalizados como e-mail, URLs e mais.

```
CREATE EXTENSION citext;

CREATE TABLE users (
  id serial PRIMARY KEY NOT NULL,
  username citext NOT NULL UNIQUE
);

INSERT INTO users (username) VALUES ('john');

INSERT INTO users (username) VALUES ('JOHN');
ERROR:  23505: duplicate key value violates unique constraint "users_username_key"
DETAIL:  Key (username)=(JOHN) already exists.
```

http://fnando.me/1ec

```
CREATE EXTENSION emailaddr;

CREATE TABLE users (
  id serial PRIMARY KEY NOT NULL,
  email emailaddr NOT NULL UNIQUE
);

INSERT INTO users (email) VALUES ('john');
ERROR:  22P02: invalid input syntax for type emailaddr: missing "@"
```

http://fnando.me/1ed

```
CREATE EXTENSION uri;

CREATE TABLE users (
  id serial PRIMARY KEY NOT NULL,
  site uri NOT NULL CHECK (
          uri_scheme(site::uri) IS NOT NULL
      AND uri_scheme(site::uri) ~ '^https?$')
);


INSERT INTO users (site) VALUES ('http://example.com');


INSERT INTO users (site) VALUES ('invalid');
ERROR:  23514: new row for relation "users" violates check constraint "users_site_check"
DETAIL:  Failing row contains (2, invalid).
```

http://fnando.me/1ee

GARANTA A INTEGRIDADE DE DADOS

Com o PostgreSQL você pode garantir a integridade dos dados com regras personalizadas.

USE CHECKS PARA VALIDAR O FORMATO DOS DADOS

Use constraints para garantir a integridade dos dados antes de persistir as informações.

```
CREATE TABLE events (
  id serial NOT NULL PRIMARY KEY,
  starts_at timestamp NOT NULL,
  ends_at timestamp NOT NULL CHECK (ends_at >= starts_at)
);


INSERT INTO events (starts_at, ends_at)
            VALUES ('2015-11-28'::timestamp, '2015-11-29'::timestamp);


INSERT INTO events (starts_at, ends_at)
            VALUES ('2015-11-28'::timestamp, '2015-11-27'::timestamp);

ERROR:  23514: new row for relation "events" violates check constraint "events_check"
DETAIL:  Failing row contains (2, 2015-11-28 00:00:00, 2015-11-27 00:00:00).
```

Apenas constraints de banco de dados podem garantir a integridade das informações.

Validações do seu framework são meramente informativas.

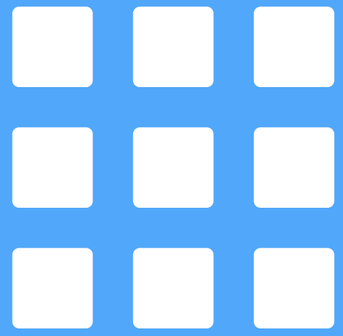
ACEITE QUE DUPLICAÇÃO DE REGRAS IRÁ EXISTIR

Ao garantir a integridade de seus dados, você acabará com regras duplicadas para informar o usuário. E isso não é um problema.

```sql
CREATE TABLE events (
  id serial NOT NULL PRIMARY KEY,
  starts_at timestamp NOT NULL,
  ends_at timestamp NOT NULL CHECK (ends_at > starts_at)
);
```

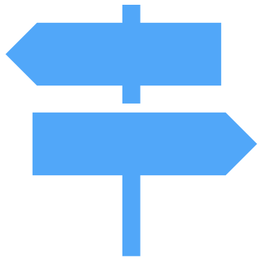
```ruby
class Event < ActiveRecord::Base
  validates_datetime :ends_at, after: :starts_at
end
```

http://fnando.me/1e4



ORGANIZE SUAS QUERIES COM CTES

SQL não é das linguagens mais legíveis do mundo, mas você pode aumentar a legibilidade com Common Table Expressions.



CTES DEFINEM A NARRATIVA DAS QUERIES

Queries complexas podem ser quebradas em pedaços menores e mais simples de entender.

```sql
CREATE TABLE countries (
  id serial PRIMARY KEY NOT NULL,
  name text NOT NULL,
  population bigint NOT NULL DEFAULT 0
);

SELECT name FROM countries
WHERE population > (SELECT population FROM countries WHERE name = 'Brazil')
ORDER BY name ASC;
```

```
     name
--------------
 China
 India
 United States
 Indonesia
(4 rows)
```

Exibir países mais populosos que o Brasil.

```sql
WITH brazil AS (
  SELECT population
  FROM countries
  WHERE name = 'Brazil'
)

SELECT name
FROM countries, brazil
WHERE countries.population > brazil.population
ORDER BY countries.population DESC;
```

```
     name
--------------
 China
 India
 United States
 Indonesia
(4 rows)
```

Exibir países mais populosos que o Brasil.

SUPORTE PARA FULL-TEXT SEARCH

O suporte para full-text search do PostgreSQL é suficiente e muito melhor que consultas com LIKE.

Similaridade

Configurável

Stemming

Sinônimos

Idiomas

Snowball

Stop Words

Ranking

Thesaurus

O Rails possui integração facilitada com full-text search do PostgreSQL através da gem pg_search.

http://fnando.me/1eb

AGREGAÇÃO COM FILTROS CONDICIONAIS

O PostgreSQL permite escrever filtros sem a necessidade de usar hacks.

```sql
SELECT
  COUNT(id) AS total,
  SUM(CASE WHEN deleted_at IS NOT NULL THEN 1 ELSE 0 END) AS deleted_accounts,
  SUM(CASE WHEN activated_at IS NOT NULL THEN 1 ELSE 0 END) AS activated_accounts
FROM users;
```

```
 total | deleted_accounts | activated_accounts
-------+------------------+--------------------
  3000 |              311 |               2266
(1 row)
```

Exibir a quantidade de usuários total, removidos e que ativaram a conta.

```sql
SELECT
  COUNT(id) AS total,
  COUNT(id) FILTER (WHERE deleted_at IS NOT NULL) AS deleted_accounts,
  COUNT(id) FILTER (WHERE activated_at IS NOT NULL) AS activated_accounts
FROM users;
```

```
 total | deleted_accounts | activated_accounts
-------+------------------+--------------------
  3000 |              311 |               2266
(1 row)
```

Exibir a quantidade de usuários total, removidos e que ativaram a conta.

SUPORTE PARA VIEWS COM CACHING

O PostgreSQL tem suporte para views que persistem os resultados como se fossem tabelas.

```sql
CREATE MATERIALIZED VIEW subscription_stats AS
WITH calendar AS (
  SELECT day::date
  FROM generate_series(
    current_date - interval '1 month',
    current_date,
    INTERVAL '1 day'
  ) day
)

SELECT
  day,
  count(id) AS count
FROM calendar
LEFT JOIN users ON created_at::date = day
WHERE
  date_trunc('month', day) = date_trunc('month',  current_date)
GROUP BY day
ORDER BY day;
```

Quantidade de cadastros por dia nos últimos 30 dias

```
SELECT * FROM subscription_stats
ORDER BY day DESC
LIMIT 1;
    day     | count
------------+-------
 2015-11-26 |   250
(1 row)

INSERT INTO users (created_at) VALUES (current_date);
SELECT * FROM subscription_stats
ORDER BY day DESC
LIMIT 1;

    day     | count
------------+-------
 2015-11-26 |   250
(1 row)
```

Quantidade de cadastros por dia nos últimos 30 dias

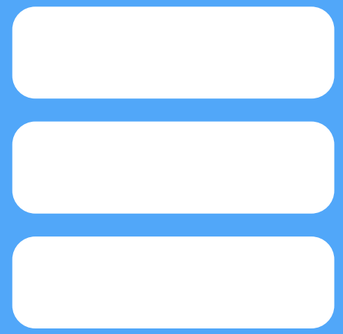
```
REFRESH MATERIALIZED VIEW subscription_stats;

SELECT * FROM subscription_stats
ORDER BY day DESC
LIMIT 1;
```

```
    day     | count
------------+-------
 2015-11-26 |   251
(1 row)
```

Quantidade de cadastros por dia nos últimos 30 dias

```ruby
class SubscriptionStats < ActiveRecord::Base
  def self.refresh
    connection.execute "REFRESH MATERIALIZED VIEW subscription_stats"
  end
end

SubscriptionStats.order(day: :desc).take(1)
[#<SubscriptionStats day: "2015-11-26", count: 251>]

User.create!
SubscriptionStats.refresh

SubscriptionStats.order(day: :desc).take(1)
[#<SubscriptionStats day: "2015-11-26", count: 252>]
```

Usando materialized views no ActiveRecord



SUPORTE PARA WINDOW FUNCTIONS

Uma maneira simples de computar cálculos em diversos registros relacionados a um grupo.

```
SELECT * FROM employees;

 id | name | salary  | department
----+------+---------+------------
  1 | John | 4500.00 | accounting
  2 | Mary | 5600.00 | sales
  3 | Paul | 7700.00 | marketing
  4 | Jane | 8400.00 | accounting
  5 | Mark | 7400.00 | sales
(5 rows)
```

Listar os salários

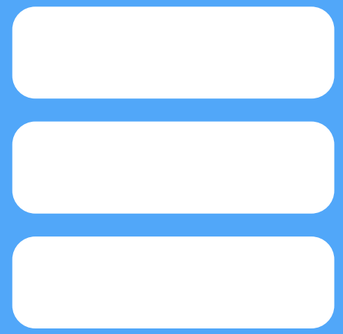
```sql
SELECT
  employees.*,
  rank() OVER (PARTITION BY department ORDER BY salary DESC)
FROM employees;
```

```
 id | name | salary  | department | rank
----+------+---------+------------+------
  4 | Jane | 8400.00 | accounting |    1
  1 | John | 4500.00 | accounting |    2
  3 | Paul | 7700.00 | marketing  |    1
  5 | Mark | 7400.00 | sales      |    1
  2 | Mary | 5600.00 | sales      |    2
(5 rows)
```

Listar os salários, ordenando pelo ranking no departamento

```
SELECT
  created_at::date,
  count(id),
  sum(count(id)) OVER (ORDER BY created_at::date) AS acc_count
FROM users
WHERE created_at > current_date - interval '1 week'
GROUP by created_at::date;

 created_at | count | acc_count
------------+-------+-----------
 2015-11-19 |   154 |       154
 2015-11-20 |   143 |       297
 2015-11-21 |   152 |       449
 2015-11-22 |   140 |       589
 2015-11-23 |   125 |       714
 2015-11-24 |   126 |       840
 2015-11-25 |   160 |      1000
(7 rows)
```

Exibir quantidade de cadastros na última semana



CRIE UMA SÉRIE DE VALORES

No PostgreSQL é muito fácil criar séries sem ter que recorrer a loops ou number tables.

```sql
SELECT number FROM generate_series(1, 10) number;

SELECT number FROM generate_series(1, 10, 2) number;

SELECT chr(generate_series(48, 57)) letter
UNION
SELECT chr(generate_series(65, 90)) letter
ORDER BY letter ASC;


SELECT day::date
FROM generate_series('2015-01-01'::date, '2015-01-31'::date, INTERVAL '1 day') day;
```

Gerando séries

```sql
SELECT
  created_at::date AS day,
  count(id) AS count
FROM articles
WHERE date_trunc('month', created_at) = date_trunc('month',  current_date)
GROUP by created_at::date
ORDER BY day;
```

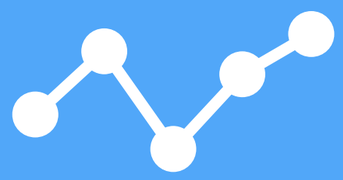
```
    day     | count
------------+-------
 2015-11-02 |     1
 2015-11-04 |     2
 2015-11-05 |     1
 ...
 2015-11-17 |     1
 2015-11-23 |     1
(11 rows)
```

Exibir quantidade de artigos criados por dia (não contínuo)

```sql
WITH calendar AS (
  SELECT day::date
  FROM generate_series(
    current_date - interval '1 month',
    current_date,
    INTERVAL '1 day'
  ) day
)

SELECT
  day,
  count(id) AS count
FROM calendar
LEFT JOIN articles ON created_at::date = day
WHERE
  date_trunc('month', day) = date_trunc('month',  current_date)
GROUP BY day
ORDER BY day;
```

```
    day      | count
-------------+-------
 2015-11-01 |     0
 2015-11-02 |     1
 2015-11-03 |     0
 2015-11-04 |     2
 2015-11-05 |     1
 2015-11-06 |     0
 2015-11-07 |     2
 2015-11-08 |     1
 2015-11-09 |     1
 2015-11-10 |     1
 2015-11-11 |     1
 ...
 2015-11-23 |     1
 2015-11-24 |     0
 2015-11-25 |     0
 2015-11-26 |     0
(26 rows)
```

Exibir quantidade de artigos criados por dia (contínuo)



INSPECIONE SEU BANCO DE DADOS

No PostgreSQL é fácil saber o que está acontecendo no seu banco de dados neste momento.

INFORMAÇÃO SOBRE TODAS AS CONEXÕES

A tabela *pg_stat_activity* retorna informações sobre todas as conexões abertas no momento.

```
SELECT * FROM pg_stat_activity;

-[ RECORD 1 ]----+--------------------------------
datid            | 812156
datname          | test
pid              | 50661
usesysid         | 10
usename          | fnando
application_name | psql
client_addr      | [NULL]
client_hostname  | [NULL]
client_port      | -1
backend_start    | 2015-11-26 09:45:40.544477-02
xact_start       | 2015-11-26 16:26:30.917594-02
query_start      | 2015-11-26 16:26:30.917594-02
state_change     | 2015-11-26 16:26:30.917644-02
waiting          | f
state            | active
backend_xid      | [NULL]
backend_xmin     | 2222027
query            | SELECT * FROM pg_stat_activity;
```

LISTA DE QUERIES MAIS LENTAS

A extensão *pg_stat_statements* retorna informações sobre as queries executadas.

```
# /etc/postgresql/9.4/main/postgresql.conf
shared_preload_libraries = 'pg_stat_statements'
pg_stat_statements.max = 10000
pg_stat_statements.track = all

CREATE EXTENSION pg_stat_statements;

SELECT pg_stat_statements_reset();
```

```
SELECT
  query,
  calls,
  total_time,
  rows,
  100.0 * shared_blks_hit / nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
FROM pg_stat_statements
ORDER BY total_time DESC
LIMIT 5;

-[ RECORD 1 ]---------------------------------------------------------------
query       | UPDATE pgbench_branches SET bbalance = bbalance + ? WHERE bid = ?;
calls       | 3000
total_time  | 5390.771
rows        | 3000
hit_percent | 99.9873144741849550
```

LISTA DE LOCKS DO POSTGRESQL

Com as tabelas *pg_stat_activity* e *pg_locks* é possível saber quais locks estão em uso.

```
CREATE VIEW pg_current_locks AS (
  SELECT blocked_locks.pid                   AS blocked_pid,
         blocked_activity.usename            AS blocked_user,
         blocking_locks.pid                  AS blocking_pid,
         blocking_activity.usename           AS blocking_user,
         blocked_activity.query              AS blocked_statement,
         blocking_activity.query             AS current_statement_in_blocking_process,
         blocked_activity.application_name   AS blocked_application,
         blocking_activity.application_name AS blocking_application
  FROM pg_catalog.pg_locks blocked_locks
  JOIN pg_catalog.pg_stat_activity blocked_activity
    ON blocked_activity.pid = blocked_locks.pid
  JOIN pg_catalog.pg_locks blocking_locks
    ON  blocking_locks.locktype = blocked_locks.locktype
    AND blocking_locks.DATABASE IS NOT DISTINCT FROM blocked_locks.DATABASE
    AND blocking_locks.relation IS NOT DISTINCT FROM blocked_locks.relation
    AND blocking_locks.page IS NOT DISTINCT FROM blocked_locks.page
    AND blocking_locks.tuple IS NOT DISTINCT FROM blocked_locks.tuple
    AND blocking_locks.virtualxid IS NOT DISTINCT FROM blocked_locks.virtualxid
    AND blocking_locks.transactionid IS NOT DISTINCT FROM blocked_locks.transactionid
    AND blocking_locks.classid IS NOT DISTINCT FROM blocked_locks.classid
    AND blocking_locks.objid IS NOT DISTINCT FROM blocked_locks.objid
    AND blocking_locks.objsubid IS NOT DISTINCT FROM blocked_locks.objsubid
    AND blocking_locks.pid != blocked_locks.pid
  JOIN pg_catalog.pg_stat_activity blocking_activity ON blocking_activity.pid = blocking_locks.pid
  WHERE NOT blocked_locks.GRANTED
);
```

http://fnando.me/1ef

_terminal 1

```
BEGIN;
SELECT * FROM users WHERE id = 1 FOR UPDATE;
```

_terminal 2

```
UPDATE users SET site = 'http://example.com' WHERE id = 1;
=> Waiting on transaction
```

_terminal 1

```
SELECT * FROM pg_current_locks;
=> View locks
```

```
-[ RECORD 1 ]-----------------------+-----------------------------------------------------
blocked_pid                         | 33782
blocked_user                        | fnando
blocking_pid                        | 34236
blocking_user                       | fnando
blocked_statement                   | update users set site = 'http://example.com' where id = 1;
current_statement_in_blocking_process | select * from users where id = 1 for update;
blocked_application                 | psql
blocking_application                | psql
```

_terminal 1

```
SELECT pg_terminate_backend(34236);
SELECT pg_cancel_backend(34236);

FATAL:  57P01: terminating connection due to administrator command
LOCATION:  ProcessInterrupts, postgres.c:2872
server closed the connection unexpectedly
   This probably means the server terminated abnormally
   before or while processing the request.
The connection to the server was lost. Attempting reset: Succeeded.

SELECT * FROM pg_current_locks;
=> 0 rows
```

O QUE VEM POR AÍ NO MUNDO DE POSTGRESQL

Veja algumas das novas funcionalidades que entrarão nas próximas versões do PostgreSQL.

POSTGRESQL COM SUPORTE A UPSERT

A única feature que o MySQL tinha e o PostgreSQL não. Estará disponível na versão 9.5.

```sql
CREATE TABLE visits (
  id serial PRIMARY KEY NOT NULL,
  url text NOT NULL UNIQUE,
  count integer NOT NULL
);

INSERT INTO visits (url, count)
            VALUES ('http://example.com', 1)
            ON CONFLICT (url)
            DO UPDATE SET count = visits.count + 1;

SELECT * FROM visits;
```

```
 id |        url         | count
----+--------------------+-------
  6 | http://example.com |     1
(1 row)
```

http://fnando.me/1ea

```
INSERT INTO visits (url, count)
            VALUES ('http://example.com', 1)
            ON CONFLICT (url)
            DO UPDATE SET count = visits.count + 1;

SELECT * FROM visits;
 id |        url         | count
----+--------------------+-------
  6 | http://example.com |     2
(1 row)
```

PARALLEL SEQUENTIAL SCAN

Workers irão executar a consulta de forma paralela. Estará disponível na versão 9.6.

```
SELECT * FROM pgbench_accounts WHERE filler LIKE '%a%';
Time: 743.061 ms

set max_parallel_degree = 4;

SELECT * FROM pgbench_accounts WHERE filler LIKE '%a%';
Time: 213.412 ms
```

http://fnando.me/1e8

AINDA É CEDO, PODE RESUMIR PARA MIM?

tl;dw

1. USE POSTGRESQL EM NOVOS PROJETOS

Não faz sentido começar novos projetos em um banco de dados obsoleto como o MySQL.

2. USE SQL MODERNO EM SEUS PROJETOS

Felizmente o PostgreSQL é bem adiantado em relação às novas versões da especificação do SQL.

https://modern-sql.com


LATERAL JOIN
GIN index
LAG
partial index
BRIN index
DISTINCT ON
savepoint
expression index
Streaming Replication
PubSub
postGIS
Foreign-Data Wrapper
PITR
GiST index
pgcrypto

Josh Berkus
@fuzzychef

I want to thank Oracle for their recent efforts to boost PostgreSQL adoption. Keep up the good work, guys!

https://twitter.com/fuzzychef/status/631222909617311744

Josh Berkus
@fuzzychef

Eu quero agradecer a Oracle pelos seus esforços recentes para aumentar a adoção do PostgreSQL. Continuem com o bom trabalho!

https://twitter.com/fuzzychef/status/631222909617311744

OBRIGADO.

@fnando