Documentos

ISSN 1677-9274
Dezembro, 2016
151

# Utilizando o formato JSON para armazenar dados do sistema BDGF no PostgreSQL


Embrapa

ISSN 1677-9274
Dezembro, 2016

*Empresa Brasileira de Pesquisa Agropecuária*
*Embrapa Informática Agropecuária*
*Ministério da Agricultura, Pecuária e Abastecimento*

# *Documentos 151*

# Utilizando o formato JSON para armazenar dados do sistema BDGF no PostgreSQL

*Fábio Danilo Vieira*

Embrapa Informática Agropecuária
Campinas, SP
2016

**Embrapa Informática Agropecuária**
Av. André Tosello, 209 - Barão Geraldo
Caixa Postal 6041 - 13083-886 - Campinas, SP
Fone: (19) 3211-5700
www.embrapa.br/informatica-agropecuaria
SAC: www.embrapa.br/fale-conosco/sac/



**1ª edição**
publicação digitalizada 2016



**Dados Internacionais de Catalogação na Publicação (CIP)**
**Embrapa Informática Agropecuária**

Vieira, Fábio Danilo.

Utilizando o formato JSON para armazenar dados do sistema BDGF no PostgreSQL / Fábio Danilo Vieira.- Campinas : Embrapa Informática Agropecuária, 2016.

23 p. : il.: cm. - (Documentos / Embrapa Informática Agropecuária, ISSN 1677-9274; 151).

1. Marcadores moleculares. 2. Diagrama entidade-relacionamento. 3. NoSQL. 4. MongoDB. 5. Java. I. Embrapa Informática Agropecuária. II. Título. III. Série.

CDD 005.74

# Autores

**Fábio Danilo Vieira**
Tecnólogo em Processamento de dados, mestre em Engenharia Agrícola, analista da Embrapa Informática Agropecuária, Campinas, SP.

# Apresentação

Nos últimos anos, a utilização da tecnologia de genotipagem em larga escala de milhares de marcadores moleculares do tipo Single Nucleotide Polymorphisms (SNP) para estimar o perfil genômico de animais permitiu o desenvolvimento de diversos estudos de associação genótipo-fenótipo em escala genômica. Entretanto, essa situação implica na necessidade de armazenamento de grande volume de dados de genotipagem, fenotipagem e pedigree de um grande número de animais, uma tendência que possivelmente aumentará ao longo dos próximos anos.

Geralmente, uma necessidade básica a todas essas ações é a utilização de uma estrutura para armazenamento dos conjuntos de dados, incluindo genótipos, fenótipos e pedigree. Dado o volume de dados considerado, uma questão importante a se considerar no desenvolvimento de uma solução computacional é a adequabilidade da modelagem do banco de dados à aplicação desejada, pois esta terá impacto direto nos tempos de consulta e escrita nos bancos de dados.

Visando ao armazenamento e à consulta eficiente desse alto volume de dados, o sistema Banco de Dados de Genótipos e Fenótipos (BDGF) foi desenvolvido utilizando um banco de dados que foi desenhado de forma que possibilitasse a implementação do tipo *JavaScript Object Notation* (JSON) em algumas tabelas do diagrama. Com a implementação do tipo JSON, é possível o uso da abordagem *Not Only SQL* (NoSQL) para armazenar parte dos dados sem que seja necessário se importar com a normalização dos mesmos.

Neste documento, será apresentada uma contribuição da Embrapa Informática Agropecuária quanto ao uso do tipo de dados JSON no PostgreSQL, sua integração com a linguagem Java e um pequeno tutorial mostrando como trabalhar com o tipo JSON no PostgreSQL.

***Silvia Maria Fonseca Silveira Massruhá***
Chefe-geral
Embrapa Informática Agropecuária

# Sumário

# Utilizando o formato JSON para armazenar dados do sistema BDGF no PostgreSQL

*Fábio Danilo Vieira*

## Introdução

Quando o diagrama entidade-relacionamento (DER) de Higa e Oliveira (2015) foi desenhado, o principal objetivo era construir um diagrama que fornecesse todos os requisitos, em termos de tabelas, índices e normalização, para armazenagem de dados de animais e seus genótipos e fenótipos associados, requisitos esses que pudessem se encaixar na maioria dos sistemas que trabalhasse com estudos de associação genótipo-fenótipo.

Entretanto, bancos de dados de genótipos e fenótipos comumente trabalham com grandes volumes de dados. Nos dias atuais, as tecnologias para geração de dados moleculares são capazes de genotipar centenas de milhares de marcadores SNP em um único ensaio para cada indivíduo, com uma grande velocidade de processamento (CAETANO, 2009). Esse imenso volume de dados, na imensa maioria das vezes, se torna um problema em sistemas gerenciadores de bancos de dados relacionais (RDBMS – do inglês *Relational Database Management System*) quando o assunto é rapidez na consulta e eficiência no armazenamento.

Por outro lado, não era desejável adotar-se uma solução totalmente NoSQL, pois a falta de experiência técnica e confiança da equipe nessa solução seria uma barreira que exigiria muito tempo e esforço para ser superada nesse momento. Dessa maneira, identificou-se que o PostgreSQL, desde a versão 9.2, vem adotando o formato JSON para a definição de campos de suas tabelas, o que permite, ao mesmo tempo, o uso de banco de dados estruturado (tabelas e suas colunas, relacionamentos, índices, etc.) com a opção NoSQL para os campos de tabelas. JSON significa *JavaScript Object Notation* (em português, Notação de Objetos JavaScript). JSON é um formato padrão aberto que consiste de pares “chave:valor”. Ao contrário de outros formatos, o JSON é um texto legível. Além disso, faz parte de um subconjunto da notação de objeto de JavaScript, porém seu uso não requer JavaScript exclusivamente.

## PostreSQL + JSON versus NoSQL

Entre as vantagens normalmente citadas para os sistemas de gerenciamento de banco de dados, NoSQL é seu desempenho, pois trabalham com estruturas de dados mais simples do que os bancos de dados SQL. Em testes realizados por (PARKER et al., 2013) utilizando um banco de dados de tamanho modesto, comparando um banco de dados relacional SQL e um banco NoSQL, os resultados mostraram que o banco NoSQL trabalhou tão bem ou melhor que o banco relacional em diversas ocasiões, exceto naquelas em que funções de agregação foram utilizadas.

No entanto, benchmarks realizados pelo site EnterpriseDB (enterprisedb.com) em 2014 mostraram que o desempenho do PostgreSQL, com o recurso JSON, foi significativamente melhor do que o do MongoDB (Figura 1). Os testes basearam-se na seleção, carregamento e inserção de dados de documentos complexos envolvendo cerca de 50 milhões de registros. O PostgreSQL foi cerca de duas vezes mais rápido na importação de dados, duas vezes e meia mais rápido na seleção e três vezes mais rápido em inserções de dados. Outro fato constatado é que o PostgreSQL consumiu 25% menos de espaço em disco (LINSTER, 2014).

Ainda assim, segundo Levy (2016), com a implementação do recurso WiredTiger, o MongoDB, em sua versão 3.0, conseguiu melhorar sua velocidade de gravação (entre 7 e 10 vezes mais rápido), assim como a

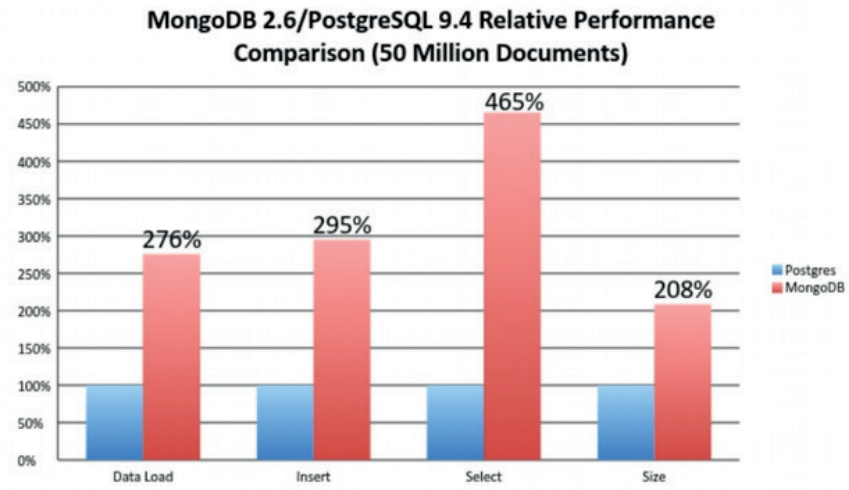


**Figura 1.** Testes de performance comparando PostgreSQL e MongoDB.
Fonte: Linster (2014).

compressão de dados, que atingiu cerca de 50% de redução do espaço em disco.

A escolha entre um banco de dados relacional como o PostgreSQL, com recursos JSON, ou um banco NoSQL, como MongoDB, para o armazenamento de dados dependerá exclusivamente dos objetivos e circunstâncias que norteiam o desenvolvimento de um sistema. Gerencidadores sofisticados de RDBMS englobam um conjunto muito rico de recursos e funcionalidades, focados principalmente no uso de Online Transaction Processing (OLTP) ou Processamento de Transações em Tempo Real, e no uso de Data Warehousing (que possibilita a análise de grandes volumes de dados coletados dos sistemas transacionais (OLTP)). Por outro lado, os sistemas NoSQL englobam um conjunto mais limitado de recursos e funcionalidades, mas fornecem escalabilidade horizontal eficiente, alta disponibilidade e flexibilidade na modelagem de dados para aplicativos específicos, os quais gerenciam dados e consultas mais simples (SAHNI; SEGLEAU, 2013).

Muitos dos recursos avançados que são comuns com sistemas RDBMS não estão presentes nos sistemas NoSQL. Isso requer que os usuários

de tecnologia NoSQL integrem seus dados com os sistemas RDBMS para fazer uso dos recursos avançados de que precisam. Os sistemas NoSQL podem fazer muitas operações simples e de propósito específico com extrema rapidez, mas não são projetados para executar operações complexas de propósito geral de forma integrada (SAHNI; SEGLEAU, 2013).

Outra análise que deve ser feita está relacionada a custos e disponibilidade das plataformas de hospedagem para PostgreSQL e MongoDB, bem como a facilidade de contratação de desenvolvedores com as habilidades correspondentes. Os recursos de conhecimento e talentos profissionais do PostgreSQL foram construídos ao longo do tempo, incentivados, entre outras coisas, pela inclusão do PostgreSQL em sistemas operacionais Linux. Por outro lado, desde sua introdução, o MongoDB já figura entre os banco de dados mais populares do mercado, sugerindo que ele também se beneficia de um pool razoável de talentos (LEVY, 2016).

Seja qual for a escolha do desenvolvedor, a implementação do recurso JSON em um banco de dados relacional (PostgreSQL), com possibilidade de superar a solução líder NoSQL, é uma conquista impressionante. E o mais importante, atingindo ótimos níveis de velocidade, eficiência e flexibilidade, que tanto as empresas exigem para aplicativos de missão crítica (LINSTER, 2014).

## Desenvolvimento Java com PostgreSQL + JSON

Para o desenvolvimento de sistemas web em linguagem Java (https://www.java.com/) utilizando o novo recurso JSON do PostgreSQL, o mapeamento de colunas JSON necessita de apenas algumas pequenas implementações no *framework* Hibernate (http://hibernate.org), quando a opção for a utilização do *framework*, obviamente. A primeira delas é a criação de uma classe que extende o *dialect* do PostgreSQL, a qual irá informar o *framework* sobre o mapeamento do tipo de dados JSON (Figura 2).

O passo seguinte é implementar a interface org.hibernate.usertype.UserType. A implementação irá mapear valores do tipo *String* para o tipo JSON e vice-versa. A Figura 3 exibe o código da implementação. O código

```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package com.jsf.util;

import java.sql.Types;
import org.hibernate.dialect.PostgreSQL9Dialect;

/**
 *
 * @author fabiodv
 */
public class JsonPostgreSQLDialect extends PostgreSQL9Dialect {

    public JsonPostgreSQLDialect() {

        super();

        this.registerColumnType(Types.JAVA_OBJECT, "json");
    }
}
```

**Figura 2.** Classe que extende PostgreSQL9Dialect para mapeamento do tipo JSON.

foi compactado para que coubesse na figura, por isso a visualização não se torna a ideal. Porém, esse e outros códigos Java aqui exibidos podem ser conferidos nesse link (http://stackoverflow.com/questions/15974474/mapping-postgresql-json-column-to-hibernate-value-type).

A próxima etapa é mais simples, que consiste na anotação das classes entidades (que mapeiam as tabelas do banco). A anotação a seguir (Figura 4) deve ser inserida na declaração de classe da entidade (acima da anotação @Entity):

O passo final é anotar a propriedade da classe entidade relacionada ao campo JSON da tabela no PosgreSQL (Figura 5):

Para trabalhar com formatação dos valores vindos em formato *String* obtidos do campo JSON, uma das opções mais utilizada é trabalhar com o pacote JSONObject (https://developer.android.com/reference/org/json/JSONObject.html), que fornece diversos métodos para mapear e manipular valores no formato JSON. Abaixo segue um pequeno trecho de código utilizado no sistema BDGF, onde se converte o campo “data”, que está no formato JSON  no banco PostgreSQL, num objeto do tipo JSONObject para manipulá-lo conforme a necessidade de implementação (Figura 6).

```
public class StringJsonUserType implements UserType {

    /** Return the SQL type codes for the columns mapped by this type ...7 linhas */
    @Override
    public int[] sqlTypes() {
        return new int[]{Types.JAVA_OBJECT};
    }

    /** The class returned by <tt>nullSafeGet()</tt> ...5 linhas */
    @Override
    public Class returnedClass() {
        return String.class;
    }

    /** Compare two instances of the class mapped by this type for persistence ...8 linhas */
    @Override
    public boolean equals(Object x, Object y) throws HibernateException {

        if (x == null) {

            return y == null;
        }

        return x.equals(y);
    }

    /** Get a hashcode for the instance, consistent with persistence "equality" ...3 linhas */
    @Override
    public int hashCode(Object x) throws HibernateException {

        return x.hashCode();
    }

    /** Retrieve an instance of the mapped class from a JDBC resultset ...12 linhas */
    @Override
    public Object nullSafeGet(ResultSet rs, String[] names, SessionImplementor session, Object owner) throws HibernateException, SQLException {
        if (rs.getString(names[0]) == null) {
            return null;
        }
        return rs.getString(names[0]);
    }
```

**Figura 3.** Implementação da interface UserType para mapeamento String-JSON e JSON--String.

```
@TypeDefs({@TypeDef(name = "StringJsonObject", typeClass = StringJsonUserType.class)})
```

**Figura 4.** Anotação para indicar mapeamento no tipo JSON.

```
@Type(type = "StringJsonObject")
private String data;
```

**Figura 5.** Anotação da propriedade relacionada ao campo JSON da tabela.

```
JSONObject dataJson;
if (individualItem.getIndividual().getData() != null) {
    dataJson = new JSONObject(individualItem.getIndividual().getData());
```

**Figura 6.** Trecho de código utilizando JSONObject para ler campo em formato JSON.

Como pode ser visto, o Java oferece um suporte bem adequado para manipulação de objetos JSON, ainda que o ideal fosse o *Hibernate* implementar nativamente esse mapeamento de campos JSON. Além disso, frameworks do tipo MVC (do inglês *Model View Controller*), que fazem a separação entre as camadas de apresentação e de aplicação, como o JavaServer Faces (JSF), poderiam também fornecer componentes que

trabalhassem nativamente com o tipo JSON, pois muitas adaptações ainda são necessárias para poder apresentar os dados JSON nos componentes desses *frameworks*.

Ainda assim, com a possibilidade do uso do campo JSON pelo PostgreSQL, a simplificação obtida nas *querys* SQL (menos *joins*, pois menos tabelas e relacionamentos foram necessários) e o menor tempo na resposta dessas consultas foram pontos de extrema importância na adoção desse tipo de dado no diagrama BDGF. Por exemplo, em outros softwares com interface web desenvolvidos pela Embrapa Informática Agropecuária (VIEIRA, 2010, 2012a, 2012b) com a funcionalidade de armazenamento de genótipos e fenótipos, e que contemplam algumas consultas básicas ao conjunto de dados moleculares (SNPs), uma consulta simples em cerca de 800 animais e 700 mil marcadores SNP cada um demora pelo menos 1 hora para ser processada. Uma consulta semelhante (utilizando a mesma máquina) feita no banco BDGF leva pouco menos de 1 minuto, pois a utilização do campo JSON retira um pouco da normalização necessária do banco tradicional, ou seja, o campo JSON não precisa se referenciar a nenhuma outra tabela, agilizando as consultas.

Outro fato interessante é que, como o formato JSON abrange um texto legível no formato “chave:valor”, é facilmente exportável por meio de classes Java (e ferramentas on-line) para outros formatos de arquivos, como por exemplo o *Comma-Separated Values* (CSV), o qual diversos softwares tradicionais de análise de dados são capazes de ler, interpretar e fornecer dados estatísticos importantes que possam ajudam em alguma tomada de decisão.

## Tipo JSON no banco BDGF

Abaixo, segue o DER atual do BDGF (Figura 7). É importante ressaltar que, devido à limitação do software para desenho do diagrama, os campos com nome “data” das tabelas que são do tipo JSON (variaveisSiexp, jsonSchema, category, individual e observation) estão sendo exibidos com o tipo “varchar”.

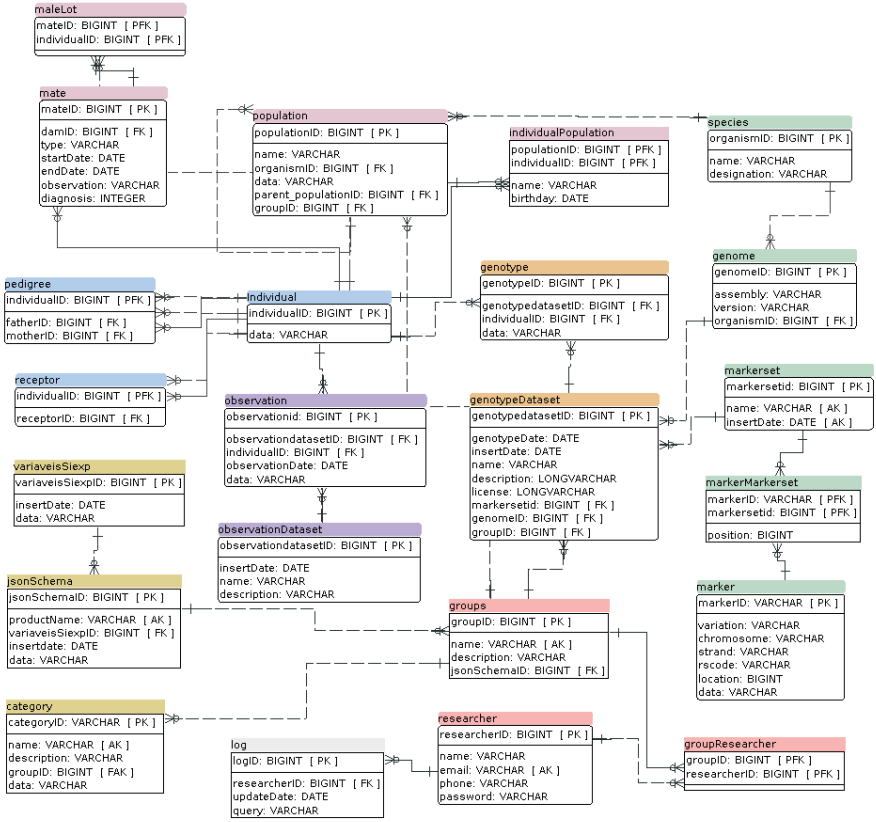


**Figura 7.** Diagrama entidade-relacionamento (DER) do BDGF.

A tabela jsonSchema (Figura 8) foi criada com o objetivo de fornecer o formato que devem seguir as variáveis de um organismo (ex.: bovino), ou seja, se devem ser do tipo inteiro, *string*, enumerada, etc. Essas variáveis são importadas do sistema de experimentos da Embrapa (SiExp), por meio de um *web-service* que tal sistema disponibiliza. O formato é então armazenado no campo “data”, com o tipo “jsonb”.

```
bdgf20160815=# \d jsonschema
                                    Table "public.jsonschema"
      Column      |       Type        |                          Modifiers
------------------+-------------------+-------------------------------------------------------------
 jsonschemaid     | bigint            | not null default nextval('jsonschema_jsonschemaid_seq'::regclass)
 productname      | character varying | not null
 variaveissiexpid | bigint            | not null
 insertdate       | date              | not null
 data             | jsonb             | not null
```

**Figura 8.** Tabela jsonSchema do banco BDGF.

As variáveis importadas do sistema de experimentos (SiExp) são armazenadas na tabela variaveisSiexp (Figura 9), também num campo chamado “data” e tipo “jsonb”. As tabelas variaveisSiexp e jsonSchema serão sempre consultadas no momento da importação dos dados de animais e de seus fenótipos. Caso uma ou mais variáveis do organismo trabalhado não esteja cadastrada na tabela variaveisSiexp, o sistema abortará a importação. E, mesmo que a variável exista na tabela variaveisSiexp, mas estiver no formato errado (verificando a tabela jsonSchema), do mesmo modo a importação será interrompida e o sistema alertará o usuário quanto ao erro.

```
bdgf20160815=# \d variaveissiexp
                                        Table "public.variaveissiexp"
      Column      |            Type             |                                 Modifiers
------------------+-----------------------------+-----------------------------------------------------------------------
 variaveissiexpid | bigint                      | not null default nextval('variaveissiexp_variaveissiexpid_seq'::regclass)
 insertdate       | timestamp without time zone | not null
 data             | jsonb                       | not null
```

**Figura 9.** Tabela variaveisSiexp do banco BDGF.

A tabela category (Figura 10) guarda as categorias das variáveis fenotípicas (tabela observation). Essas categorias podem ser, por exemplo, “Eficiência Alimentar”, “Maciez da carne”, etc., cada uma relacionada a um conjunto de características fenotípicas.

A tabela individual (Figura 11) guarda dados relacionados ao animal, também em um campo JSON. Esses dados são, em geral, variáveis fixas, ou seja, que não se alterarão com o tempo, como por exemplo, “data de nascimento”, “sexo” e “origem”.

```
bdgf20160815=# \d category
                                    Table "public.category"
   Column    |       Type        |                            Modifiers
-------------+-------------------+-------------------------------------------------------------------
 categoryid  | bigint            | not null default nextval('category_categoryid_seq'::regclass)
 name        | character varying | not null
 description | character varying |
 groupid     | bigint            | not null
 data        | jsonb             |
```

**Figura 10.** Tabela category do banco BDGF.

```
bdgf20160815=# \d individual
                                    Table "public.individual"
    Column    |            Type             |                                Modifiers
--------------+-----------------------------+---------------------------------------------------------------------
 individualid | bigint                      | not null default nextval('individual_individualid_seq'::regclass)
 data         | jsonb                       |
 insertdate   | timestamp without time zone |
```

**Figura 11.** Tabela individual do banco BDGF.

Ao contrário da tabela individual, a tabela observation (Figura 12) (relacionada a fenótipos) armazena dados que contém variáveis que indicam variações em algumas características do animal, tais como: “peso”, “altura”, “criador”, etc. Além disso, é guardada a data da observação dessas medidas no campo “observationdate”.

Colocar tipo JSON para a tabela de fenótipos se mostrou importante por diversos motivos. Um deles é que os arquivos com dados fenotípicos que serão importados podem vir com variáveis diferentes para cada animal. Um exemplo que poderia acontecer seria é o arquivo conter um animal “X” com as variáveis “peso ao nascer”, “idade desmama” e “altura” e conter um animal “Y” com as variáveis “peso ao nascer” e “cor pelagem”. Com o campo JSON, evita-se ter que criar outra tabela só para armazenar as características fenotípicas e relacioná-las à tabela observation, melhorando a performance da consulta ao evitar diversos *joins*.

```
bdgf20160815=# \d observation
                                Table "public.observation"
        Column         |  Type  |                                Modifiers
-----------------------+--------+-------------------------------------------------------------------------
 observationid         | bigint | not null default nextval('observation_observationid_seq'::regclass)
 observationdatasetid | bigint | not null
 individualid          | bigint | not null
 observationdate       | date   | not null
 data                  | jsonb  | not null
```

**Figura 12.** Tabela observation do banco BDGF.

## Conhecendo e utilizando alguns comandos JSON no PostgreSQL

Existe uma intensa documentação sobre como utilizar funções e operadores voltados ao tratamento do campo JSON na internet, especialmente no site do PostgreSQL (https://www.postgresql.org/docs/9.5/static/functions-json.html). Com o intuito de apenas mostrar uma fração mais importante deles, com a exibição de figuras ilustrando a execução dos comandos no console do PostgreSQL, essa seção foi desenvolvida.

A seguir, então, serão mostrados alguns comandos e funções úteis para lidar com o tipo JSON no PostgreSQL. Com o intuito de ser mais didático, será criada uma tabela pedidos como exemplo, a partir da qual se derivará as explicações. A Figura 13 mostra o comando para criação da tabela:

```
testejson=# CREATE TABLE pedidos (
testejson(# id serial NOT NULL PRIMARY KEY,
testejson(# info json NOT NULL
testejson(# );
CREATE TABLE
```

**Figura 13.** Criando tabela pedidos.

A tabela pedidos contém dois campos:

- id que indica a chave primária e é gerada sequencialmente pelo próprio PosgreSQL;
- info que armazena os dados no formato JSON (chave:valor).

Para inserir dados numa coluna com formato JSON, deve-se, primeiro, se certificar do formato válido (ou seja, chave:valor). A Figura 14 mostra um exemplo de como inserir uma linha de dados na coluna JSON:

```
testejson=# INSERT INTO pedidos (info) VALUES (
testejson(# '{"cliente": "Pedro Salazar", "items": {"produto":"arroz", "qtde": 10}}'
testejson(# );
INSERT 0 1
```

**Figura 14.** Inserindo uma linha de dados no campo JSON.

Como pode se observar, foi inserido o produto “arroz” para o cliente “Pedro Salazar”, e sua quantidade foi 10 (neste exemplo não foi considerado nenhuma unidade de medida, mas poderia ter um campo para isso).

Já a Figura 15 mostra como seria a inserção de múltiplas linhas de uma vez só, sem ter que realizar vários inserts.

```
testejson=# INSERT INTO pedidos (info) VALUES (
testejson(# '{"cliente": "Joao Santos", "items": {"produto":"biscoito", "qtde": 8}}'
testejson(# ),
testejson-# (
testejson(# '{"cliente": "Maria Antonieta", "items": {"produto":"pão", "qtde": 2}}'
testejson(# );
INSERT 0 2
```

**Figura 15.** Inserindo múltiplas linhas com dados JSON.

Para realizar uma consulta simples nesta tabela, basta executar um select comum, e os dados serão exibidos da seguinte forma (Figura 16):

```
testejson=# SELECT * FROM pedidos;
 id |                                 info
----+-------------------------------------------------------------------------
  1 | {"cliente": "Pedro Salazar", "items": {"produto":"arroz", "qtde": 10}}
  2 | {"cliente": "Joao Santos", "items": {"produto":"biscoito", "qtde": 8}}
  3 | {"cliente": "Maria Antonieta", "items": {"produto":"pão", "qtde": 2}}
(3 registros)
```

**Figura 16.** Consulta simples na tabela.

Como demonstrado na figura, o PosgreSQL retorna os dados no formato JSON, ou seja, chave:valor (padrão do JSON).

É possível utilizar os operadores -> e ->> para extrair valores de colunas JSON. O operador -> retorna o tipo JSON original (que pode ser um objeto), enquanto que ->> retorna o texto (Figura 17).

```
testejson=# SELECT info -> 'cliente' AS cliente FROM pedidos;
      cliente
------------------
 "Pedro Salazar"
 "Joao Santos"
 "Maria Antonieta"
(3 registros)

testejson=# SELECT info ->> 'cliente' AS cliente FROM pedidos;
     cliente
-----------------
 Pedro Salazar
 Joao Santos
 Maria Antonieta
(3 registros)
```

**Figura 17.** Trabalhando com os operadores JSON.

Também é possível usar o operador -> para retornar um objeto aninhado e assim encadear os operadores (Figura 18).

Como visto, primeiro info -> ‘items’ retorna “items” como objetos JSON, e depois info -> ‘items’ - >> ‘produto’ retorna todos os produtos como texto.

```
testejson=# SELECT info ->'items'->> 'produto' AS produto FROM pedidos;
 produto
----------
 arroz
 biscoito
 pão
(3 registros)
```

**Figura 18.** Utilizando operadores para trazer objeto aninhado (*array*).

Também é possível utilizar a cláusula *WHERE* para manipular objetos JSON. Por exemplo, é possível selecionar os clientes que compraram apenas biscoito (Figura 19):

```
testejson=# SELECT info ->> 'cliente' AS cliente FROM pedidos
WHERE info -> 'items' ->> 'produto' = 'biscoito';
   cliente
-------------
 Joao Santos
(1 registro)
```

**Figura 19.** Exemplo de uso da cláusula *WHERE* com JSON.

Neste caso, apenas o cliente “Joao Santos” comprou o produto “biscoito”.

Outras funções que podem ser utilizadas são as de agregação (min, max, average, sum), como pode ser visto na figura a seguir (Figura 20).

O formato JSON para PostgreSQL também oferece diversas funções específicas para se trabalhar com esse tipo especial de informação. Uma delas, por exemplo, informa apenas o nome das chaves do campo JSON. A Figura 21 mostra um exemplo.

```
testejson=# SELECT
testejson-#  MIN (
testejson(#  CAST (
testejson(#  info -> 'items' ->> 'qtde' AS INTEGER
testejson(#  )
testejson(#  ),
testejson-#  MAX (
testejson(#  CAST (
testejson(#  info -> 'items' ->> 'qtde' AS INTEGER
testejson(#  )
testejson(#  ),
testejson-#  SUM (
testejson(#  CAST (
testejson(#  info -> 'items' ->> 'qtde' AS INTEGER
testejson(#  )
testejson(#  ),
testejson-#  AVG (
testejson(#  CAST (
testejson(#  info -> 'items' ->> 'qtde' AS INTEGER
testejson(#  )
testejson(#  )
testejson-# FROM
testejson-#  pedidos;
 min | max | sum |        avg
-----+-----+-----+-------------------
   2 |  10 |  20 | 6.6666666666666667
(1 registro)
```

**Figura 20.** Funções de agregação usadas com JSON.

Há diversas outras funções, as quais sempre estão atualizadas e/ou incrementadas. Se digitar o comando “\df” no console do PostgreSQL, é possíve consultar todas as funções relacionadas ao tipo JSON.

```
testejson=# SELECT json_object_keys(info->'items')
testejson-# FROM pedidos;
 json_object_keys
------------------
 produto
 qtde
 produto
 qtde
 produto
 qtde
(6 registros)
```

**Figura 21.** Uso da função *json_object_keys* para obter chaves do campo JSON.

## Considerações finais

Com as versões mais recentes, o PostgreSQL iniciou uma nova era de flexibilidade para desenvolvedores, acrescentando-se como mais uma alternativa para solução de problemas relacionados a *big-data*. Com a integração do tipo JSON ao PostgreSQL, pode-se contar com todo o poder de consultas SQL com o armazenamento adequado de grandes volumes de dados sem excesso de normalizações, tornando-se uma boa solução para implementação do sistema Banco de Dados de Genótipos e Fenótipos (BDGF). Como visto, pode ser uma solução promissora para muitos problemas relacionados à performance e ao armazenamento optimizado de dados num SGBD relacional, principalmente àqueles que necessitam de um banco com modelagem estruturada ao mesmo tempo que lidam com alto volume de dados.

## Referências

CAETANO, A. R. Marcadores SNP: conceitos básicos, aplicações no manejo e no melhoramento animal e perspectivas para o futuro. **Revista Brasileira de Zootecnia**, v. 38, p. 64-71, jul. 2009. Número especial. Disponível em: <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S1516-35982009001300008&lng=en&nrm=iso>. Acesso em: 6

dez. 2016.

HIGA, R. H.; OLIVEIRA, G. B. de. **Banco de Dados de Genótipos e Fenótipos (BDGF) para suporte a estudos de associação genômica ampla e seleção genômica em programas de melhoramento animal**. Campinas: Embrapa Informática Agropecuária, 2015. 30 p. (Embrapa Informática Agropecuária. Documentos, 133).

LEVY, E. **Postgres Vs. Mongodb For Storing Json Data**: which should you choose? 2016. Disponível em: <https://www.sisense.com/blog/postgres-vs-mongodb-for-storing-json-data>. Acesso em: 6 dez. 2016.

LINSTER, M. **Postgres Outperforms MongoDB and Ushers in new developer reality**. 2014. Disponível em: <http://www.enterprisedb.com/postgres-plus-edb-blog/marc-linster/postgres-outperforms-mongodb-and-ushers-new-developer-reality>. Acesso em: 7 dez. 2016.

PARKER, Z.; POE, S.; VRBSKY, S. V. Comparing NoSQL MongoDB to an SQL DB, 2013. In Proceedings of the 51st ACM Southeast Conference, 51., 2013, Savanaah. **Proceedings**... (ACMSE '13). New York: ACM, 2013. 6 p. DOI: http://dx.doi.org/10.1145/2498328.2500047.

SAHNI, A.; SEGLEAU, D. **NoSQL and SQL Introspective Oracle NoSQL Database 11g Release 2 (11.2.1.2)**. [S.l.] Oracle Corporation, 2013.

VIEIRA, F. D. Sistema Bife de Qualidade. Versão 1.6. Campinas: Embrapa Informática Agropecuária, 2012a. 1 CD-ROM.

VIEIRA, F. D. **Sistema Consulta Dados de Ovinos**. Versão 1.0. Campinas: Embrapa Informática Agropecuária, 2010. 1 CD-ROM.

VIEIRA, F. D. **Sistema Suínos**. Versão 1.1. Campinas: Embrapa Informática Agropecuária, 2012b. 1 CD-ROM.

Embrapa

Informática Agropecuária

MINISTÉRIO DA
AGRICULTURA, PECUÁRIA
E ABASTECIMENTO

BRASIL
GOVERNO FEDERAL

CGPE 13506