

# Working With PostgreSQL as a JSON Document Store

**Robert Bernier**

Robert.bernier@percona.com

Senior PostgreSQL Consultant

Percona



# Howdy!

---

Who am I?

# Howdy!

---

Who RU?

# Make Sense of Your Database

## About This Talk

- **Part I: An Overview of PostgreSQL JSON**
- **Part II: Scenarios**

# Make Sense of Your Database

## PART I: An Overview of PostgreSQL JSON

### Some Definitions

- 1) DBMS
- 2) RDBMS
- 3) GPDBMS

# PostgreSQL: General Purpose Database Management System

- Relational (structured)
  - row wise: OLTP
  - column wise: OLAP
  - data warehouse
  - analytics
- Unstructured
  - hash
  - hierarchical
  - document storage
- High Availability (HA)
  - logical backups
  - clustered, across nodes ex: replication
- High Performance (HP)
  - 1) clustered ex: horizontal scaling
    - sharding with data redundancies
- Embedded (application centric)
- Big Data (inter/intra data centres)
- Encrypted
  - session
  - data
  - at rest (encrypted discs)

# About JSON

Unstructured data files often include text and multimedia content. Examples include e-mail messages, word processing documents, videos, photos, audio files, presentations, webpages, and many other kinds of business documents.

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999.

JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

<http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>

# About the PostgreSQL Implementation of JSON

- 1) Data Types
- 2) Indexes
- 3) JSON Path Type
- 4) Operators
- 5) Functions



# Data Types

```
create table mytable (  
  id integer primary key generated by default as identity,  
  mydoc json,  
  mydocb jsonb  
);
```

```
Table "public.mytable"  
Column | Type      | Collation | Nullable | Default  
-----+-----+-----+-----+-----  
id      | integer   |           | not null | generated by default as identity  
mydoc   | json      |           |          |  
mydocb  | jsonb     |           |          |  
Indexes:  
    "mytable_pkey" PRIMARY KEY, btree (id)
```

<https://www.postgresql.org/docs/current/datatype-json.html>

# Example: Simple Scalar/Primitive Values

-- Primitive values can be numbers, quoted strings, true, false, or null

```
SELECT '5'::json;
```

-- Array of zero or more elements (elements need not be of same type)

```
SELECT '[1, 2, "foo", null]'::json;
```

-- Object containing pairs of keys and values

-- Note that object keys must always be quoted strings

```
SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::json;
```

-- Arrays and objects can be nested arbitrarily

```
SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;
```

# Example: JSONB Containment and Existence

-- Simple scalar/primitive values contain only the identical value:

```
SELECT '"foo"'::jsonb @> '"foo"'::jsonb;
```

-- The array on the right side is contained within the one on the left:

```
SELECT '[1, 2, 3]'::jsonb @> '[1, 3]'::jsonb;
```

-- Order of array elements is not significant, so this is also true:

```
SELECT '[1, 2, 3]'::jsonb @> '[3, 1]'::jsonb;
```

-- Duplicate array elements don't matter either:

```
SELECT '[1, 2, 3]'::jsonb @> '[1, 2, 2]'::jsonb;
```

-- The object with a single pair on the right side is contained

-- within the object on the left side:

```
SELECT '{"product": "PostgreSQL", "version": 9.4, "jsonb": true}'::jsonb @> '{"version": 9.4}'::jsonb;
```

-- The array on the right side is not considered contained within the

-- array on the left, even though a similar array is nested within it:

```
SELECT '[1, 2, [1, 3]]'::jsonb @> '[1, 3]'::jsonb; -- yields false
```

-- But with a layer of nesting, it is contained:

```
SELECT '[1, 2, [1, 3]]'::jsonb @> '[[1, 3]]'::jsonb;
```

-- Similarly, containment is not reported here:

```
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"bar": "baz"}'::jsonb; -- yields false
```

-- A top-level key and an empty object is contained:

```
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"foo": {}}'::jsonb;
```

# Indexes

```
create table mytable (  
  id integer primary key generated by default as identity,  
  mydoc json,  
  mydocb jsonb  
);
```

-- only JSONB

```
create index on mytable using gin (mydocb);
```

Table "public.mytable"				
Column	Type	Collation	Nullable	Default
id	integer		not null	generated by default as identity
mydoc	json			
mydocb	jsonb			

Indexes:

```
"mytable_pkey" PRIMARY KEY, btree (id)  
"mytable_mydocb_idx" gin (mydocb)
```

# Example

```
{  
  "guid": "9c36adc1-7fb5-4d5b-83b4-90356a46061a",  
  "name": "Angela Barton",  
  "is_active": true,  
  "company": "MagnaFone",  
  "address": "178 Howard Place, Gulf, Washington, 702",  
  "registered": "2009-11-07T08:53:22 +08:00",  
  "latitude": 19.793713,  
  "longitude": 86.513373,  
  "tags": [  
    "enim",  
    "aliquip",  
    "qui"  
  ]  
}
```

# Example, cont'd

- 
- Find documents in which the key "company" has value "Magnafone"
- Does the left JSON value contain the right
- JSON path/value entries at the top level?
- 

```
tmp=# EXPLAIN
tmp=# SELECT mydocb->'guid', mydocb->'name'
tmp=# FROM mytable
tmp=# WHERE mydocb @> '{"company": "Magnafone"}';
          QUERY PLAN
```

```
-----
Seq Scan on mytable  (cost=0.00..20.63 rows=1 width=64)
  Filter: (mydocb @> '{"company": "Magnafone"}'::jsonb)
```

# Example, cont'd

```
tmp=# create index on mytable using gin ((mydocb -> 'tags'));  
--
```

```
tmp=# EXPLAIN
```

```
tmp=# SELECT mydocb->'guid', mydocb->'name'
```

```
tmp=# FROM mytable
```

```
tmp=# WHERE mydocb @> '{"company": "Magnafone"}';
```

```
QUERY PLAN
```

```
-----  
Bitmap Heap Scan on mytable (cost=12.01..16.02 rows=1 width=64)
```

```
  Recheck Cond: (mydocb @> '{"company": "Magnafone"}'::jsonb)
```

```
    -> Bitmap Index Scan on mytable_mydocb_idx (cost=0.00..12.01 rows=1 width=0)
```

```
        Index Cond: (mydocb @> '{"company": "Magnafone"}'::jsonb)
```

# JSON Path Type

The `jsonpath` type implements support for the SQL/JSON path language.

Semantics:

- Dot (.) is used for member access.
- Square brackets ([]) are used for array access.
- SQL/JSON arrays are 0-relative, unlike regular SQL arrays that start from 1.

<https://www.postgresql.org/docs/current/datatype-json.html>

<https://www.postgresql.org/docs/current/functions-json.html>



# Operators

EXAMPLE: (Operator ->)

Get JSON array element

*(indexed from zero, negative integers count from the end)*

```
SELECT '["a":"foo"],["b":"bar"],["c":"baz"]'::json->2 as "operator ->";
```

```
operator ->
```

```
-----
```

```
{"c":"baz"}
```

# Operators: JSON and JSONB

Operator	Example	Example Result
->	'[{"a":"foo"}, {"b":"bar"}, {"c":"baz"}]'::json->2	{ "c": "baz" }
->	'{"a": {"b":"foo"}}'::json->'a'	{ "b": "foo" }
->>	'[1,2,3]'::json->>2	3
->>	'{"a":1, "b":2}'::json->>'b'	2
#>	'{"a": {"b":{"c": "foo"}}}'::json#>'{a,b}'	{ "c": "foo" }
#>>	'{"a":[1,2,3], "b":[4,5,6]}'::json#>>'{a,2}'	3
@>	'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb	t
<@	'{"b":2}'::jsonb <@ '{"a":1, "b":2}'::jsonb	t
?	'{"a":1, "b":2}'::jsonb ? 'b'	t
?	'{"a":1, "b":2, "c":3}'::jsonb ?  array['b', 'c']	t
?&	'["a", "b"]'::jsonb ?& array['a', 'b']	t
	'["a", "b"]'::jsonb    '["c", "d"]'::jsonb	["a", "b", "c", "d"]
-	'{"a": "b"}'::jsonb - 'a'	{}
-	'{"a": "b", "c": "d"}'::jsonb - '{a,c}'::text[]	{}
-	'["a", "b"]'::jsonb - 1	["a"]
#-	'["a", {"b":1}]'::jsonb #- '{1,b}'	["a", {}]
@?	'{"a":[1,2,3,4,5]}'::jsonb @? '\$.a[*] ? (@ > 2)'	t
@@	'{"a":[1,2,3,4,5]}'::jsonb @@ '\$.a[*] > 2'	t

# Operators: JSONB only

## Operator Details

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
=	equal
<> or !=	not equal

# Functions

```
SELECT to_json('Fred said "Hi." '::text);
       to_json
```

```
-----
"Fred said \"Hi.\""
```

```
SELECT array_to_json('{{1,5},{99,100}}'::int[]);
       array_to_json
```

```
-----
[[1,5],[99,100]]
```

```
SELECT * FROM json_each('{ "a": "foo", "b": "bar" }');
```

key	value
a	"foo"
b	"bar"

# Functions Cont'd

```
row_to_json(row(1,'foo'))  
json_build_array(1,2,'3',4,5)  
json_build_object('foo',1,'bar',2)  
json_object('{a, 1, b, "def", c, 3.5}') json_object('{{a, 1},{b, "def"},{c, 3.5}}')  
json_object('{a, b}', '{1,2}')  
json_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4]')
```

```
{"f1":1,"f2":"foo"}  
[1, 2, "3", 4, 5]  
{"foo": 1, "bar": 2}  
{"a": "1", "b": "def", "c": "3.5"}  
{"a": "1", "b": "2"}  
5
```

# Functions Cont'd

```
json_extract_path(from_json json, VARIADIC path_elems text[]) jsonb_extract_path(from_json jsonb, VARIADIC path_elems text[])
json_extract_path_text(from_json json, VARIADIC path_elems text[]) jsonb_extract_path_text(from_json jsonb, VARIADIC path_elems text[])
json_object_keys(json) jsonb_object_keys(jsonb)
json_populate_record(base anyelement, from_json json) jsonb_populate_record(base anyelement, from_json jsonb)
json_populate_recordset(base anyelement, from_json json) jsonb_populate_recordset(base anyelement, from_json jsonb)
json_array_elements(json) jsonb_array_elements(jsonb)
json_array_elements_text(json) jsonb_array_elements_text(jsonb)
json_typeof(json) jsonb_typeof(jsonb)
json_to_record(json) jsonb_to_record(jsonb)
json_to_recordset(json) jsonb_to_recordset(jsonb)
json_strip_nulls(from_json json) jsonb_strip_nulls(from_json jsonb)
jsonb_set(target jsonb, path text[], new_value jsonb [, create_missing boolean])
jsonb_insert(target jsonb, path text[], new_value jsonb [, insert_after boolean])
```

# Make Sense of Your Database

## Part II: Scenarios

### Create Database

- Populate Database
- Translate Relational to JSON
- Queries

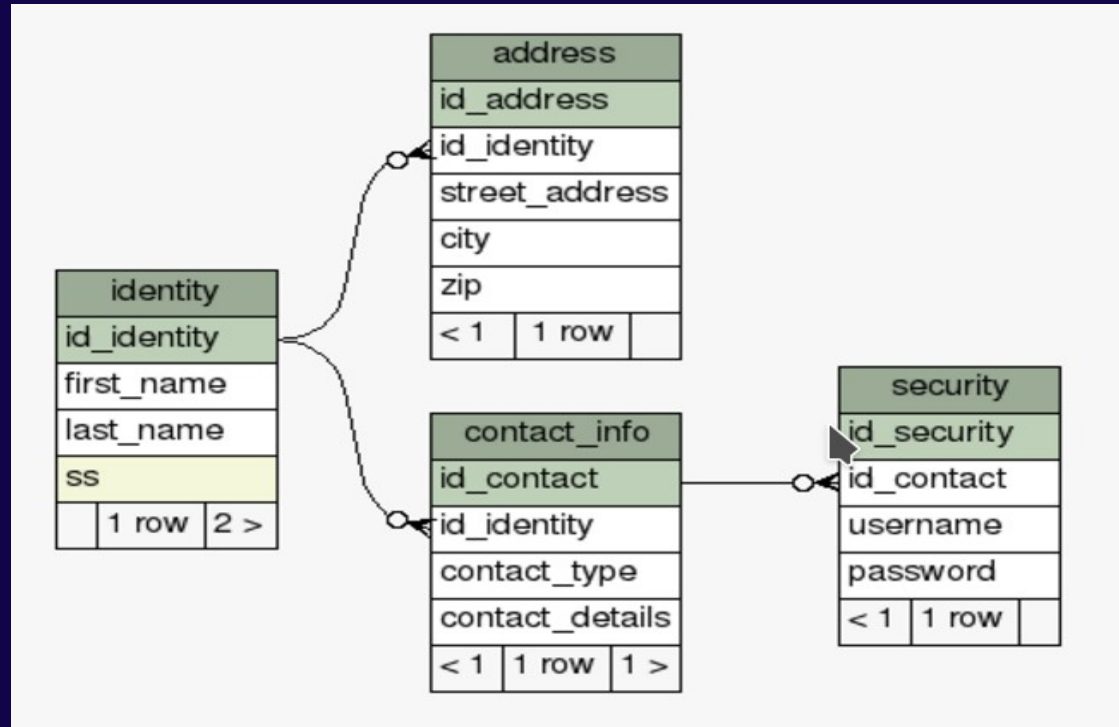
# Create Database

```
BEGIN;  
create extension pgcrypto;  
create or replace function f_password_hash()  
returns trigger  
as  
$$  
begin  
    NEW.password = crypt(NEW.password::text, gen_salt('md5'));  
    RETURN NEW;  
end;  
$$  
language plpgsql;  
  
create type type_communication as enum ('email', 'slack', 'skype');  
  
create table if not exists identity (  
    id_identity integer primary key generated by default as identity,  
    first_name varchar(15) not null,  
    last_name varchar(60) not null,  
    ss char(9) unique not null check (ss ~ '^[[:digit:]]{9}$')  
);  
  
create table if not exists address (  
    id_address integer primary key generated by default as identity,  
    id_identity integer references identity(id_identity) not null ,  
    street_address varchar(128) null,  
    city varchar(64) null,  
    zip char(5) null check (zip ~ '^[[:digit:]]{5}$')  
);  
  
create table if not exists contact_info (  
    id_contact integer primary key generated by default as identity,  
    id_identity integer references identity(id_identity) not null,  
    contact_type type_communication not null,  
    contact_details varchar(64) not null  
);
```

```
create table if not exists security (  
    id_security integer primary key generated by default as identity,  
    id_contact integer references contact_info (id_contact),  
    username varchar(16) not null,  
    password varchar not null  
);  
  
create table document (  
    id_doc integer primary key generated by default as identity,  
    myrecord json  
);  
  
create trigger tr_password_hash  
before insert or update  
on security  
for each row  
execute function f_password_hash();  
COMMIT;
```



# Entity Relationship Diagram



# Populate Database

```
BEGIN;
```

```
insert into identity(first_name,last_name,ss)
  values ('robert','bernier',123456789) returning id_identity;
```

```
insert into address (id_identity, street_address, city, zip)
  values (1,'123 my address','my city','12345') returning id_address;
```

```
insert into contact_info (id_identity, contact_type, contact_details)
  values (1, 'email','robert.bernier@percona.com'),
  (1, 'skype','rbernier_zulu'),
  (1, 'slack','rbernier') returning id_contact;
```

```
insert into security (id_contact, username, password)
  values (1,'user1','mypassword'),
  (2,'user2','mypassword'),
  (3,'user3','mypassword')
  returning id_security, password;
```

```
insert into identity(first_name,last_name,ss)
  values ('conrad','black',234567891) returning id_identity;
```

```
insert into address (id_identity, street_address, city, zip)
  values (2,'1313 mocking bird lane','smallville','23451')
  returning id_address;
```

```
insert into contact_info (id_identity, contact_type, contact_details)
  values (2, 'email','user1'),
  (2, 'skype','user2') returning id_contact;
```

```
COMMIT;
ANALYZE;
```

# Translate Relational to JSON

## row\_to\_json

```
-----  
\df row_to_json  
  
                List of functions  
 Schema      | Name      | Result data type | Argument data types | Type  
-----+-----+-----+-----+-----  
 pg_catalog  | row_to_json | json              | record               | func  
 pg_catalog  | row_to_json | json              | record, boolean     | func
```

# Translate Relational to JSON

## EX: row\_to\_json

```
with a (i,j,k,l,m,n,o) as
(
  select id_identity,
         first_name,
         last_name,
         ss,
         street_address,
         city m,
         zip n
  from identity
  join address using (id_identity)
),
b as (select row_to_json(row(i,j,k,l,m,n,o)) as r from a)
select * from b;
```

r

---

```
{"f1":1,"f2":"robert","f3":"bernier","f4":"123456789","f5":"123 my address","f6":"my city","f7":"12345"}
{"f1":2,"f2":"conrad","f3":"black","f4":"234567891","f5":"1313 mocking bird lane","f6":"smallville","f7":"23451"}
```

# Translate Relational to JSON

## EX cont'd: row\_to\_json

with a (i,j,k) as

```
(
  select id_identity,
         username,
         password
  from security
  join contact_info using (id_contact)
  join identity using (id_identity)
)
select row_to_json(row(i,j,k)) as r from a;
```

r

```
-----
{"f1":1,"f2":"user1","f3":"$1$alvLhvAx$iseOfgiKDbvRrWRPz9Jpb/" }
{"f1":1,"f2":"user2","f3":"$1$VADRCU8H$U5YGB7NNJWM9vdSe9QJah." }
{"f1":1,"f2":"user3","f3":"$1$ko4Yb/wR$j.6/eEIcuJ.4rAEL8rpuo1" }
```

# Translate Relational to JSON

## EX: Building The Document

with

```
a as (select id_identity,
            username,
            password
      from security
      join contact_info using (id_contact)
      join identity using (id_identity)),
b as (select json_agg(json_build_object(username,password)) from a),
c as (select json_build_object('username_password',json_agg) as charlie from b),
d as (select json_build_object (
      'id_identity', id_identity,
      'first_name', first_name,
      'last_name', last_name,
      'ss', ss,
      'street_address',street_address,
      'city', city,
      'zip', zip
    ) as delta from identity
      join address using (id_identity))
insert into document(myrecord)
select delta::jsonb || charlie::jsonb from c,d;
```

# Queries

```
-- QUERY 1:  
select myrecord from document;
```

```
{  
  "ss": "123456789", "zip": "12345", "city": "my city", "last_name": "bernier", "first_name":  
  "robert", "id_identity": 1, "street_address": "123 my address", "username_password":  
  [{  
    "robert.bernier": "$1$LZoB/slh$W.stf.vWGrR7/diNJ/UeU1",  
    "rbernier_zulu": "$1$uGnN/7tV  
$FiFem5POZKILYnGAjNutm0",  
    "rbernier": "$1$MPPrHGpH$dqeVGLSAl2cT2CS44MTUa0",  
    "conrad.black":  
    "$1$6uDGVVu9$qIKVZht3EYjdjIIKT1YX00",  
    "cblack_literary": "$1$dibFFjBx$23/xMXavCVc0IscDPM00f0"}  
  ]  
}
```

# Queries

```
-- QUERY 2:  
select myrecord -> 'first_name' as "first name" from document limit 1;  
  
first name  
-----  
"robert"
```



# Queries

```
-- QUERY 3:  
select myrecord -> 'last_name' as "last name" from document limit 1;  
  
last name  
-----  
"bernier"
```

# Queries

```
-- QUERY 4:  
select myrecord -> 'username_password' as "username/passsswords" from document limit 1;  
-----  
[{"robert.bernier": "$1$AKXaaSQM$0029cNu0PHb/7UdoF/CbV1"}, {"rbernier_zulu":  
"$1$arVZSMo3$UVCjAVhyhXQmfpnYDkpIY1"}, {"rbernier": "$1$Q7urUIj0$Sm992oUd7Y4dc/QruSV7Z0"}]
```

# Queries

```
-- QUERY 5:
select myrecord -> 'username_password' ->> 0 as "username/password" from document limit 1;

username/password
-----
{"robert.bernier": "$1$AKXaaSQM$0029cNu0PHb/7UdoF/CbV1"}
```

# Queries

```
-- QUERY 6:  
select myrecord -> 'username_password' ->> 1 as "username/password" from document limit 1;  
  
username/password  
-----  
{"rbernier_zulu": "$1$arVZSMo3$UVCjAVhyhXQmfpnYDkpIY1"}
```

# Queries

```
-- QUERY 7:
select myrecord -> 'username_password' ->> 2 as "username/password" from document limit 1;

| | | username/password
-----
{"rbernier": "$1$Q7urUIj0$Sm992oUd7Y4dc/QruSV7Z0"}
```

# Queries

```
-- QUERY 8:  
select (myrecord -> 'username_password' ->> 0)::json -> 'robert.bernier' as "my password" from  
document limit 1;  
  
-----  
my password  
  
"$1$AKXaaSQM$0029cNu0PHb/7UdoF/CbV1"
```

# Queries

```
-- QUERY 9:
select myrecord -> 'first_name' as "first name",
       myrecord -> 'last_name' as "last name",
       myrecord -> 'street_address' as "street address",
       myrecord -> 'city' as "city",
       myrecord -> 'zip' as "zip"
from document;
```

first name	last name	street address	city	zip
"robert"	"bernier"	"123 my address"	"my city"	"12345"
"conrad"	"black"	"1313 mocking bird lane"	"smallville"	"23451"

# Queries

```
-- QUERY 10:
select myrecord -> 'first_name' as "first name",
       myrecord -> 'last_name' as "last name",
       myrecord -> 'street_address' as "street address",
       myrecord -> 'city' as "city",
       myrecord -> 'zip' as "zip"
from document
where
  myrecord ->> 'first_name' = 'conrad'
and myrecord ->> 'last_name' = 'black';
```

first name	last name	street address	city	zip
"conrad"	"black"	"1313 mocking bird lane"	"smallville"	"23451"



# Questions?

# Thank You!

[https://github.com/rbernierZulu/pg\\_conf\\_Seattle-2024](https://github.com/rbernierZulu/pg_conf_Seattle-2024)