The road to SQL/JSON features

Álvaro Herrera
PostgreSQL developer

Vancouver, Canada
29th May 2024

A Summer of Code

- Google Summer of Code project, 2010
- Introduced initial JSON support in PostgreSQL
- authored by Joseph Adams
  - Mentored by Andrew Dunstan
  - Committed by Robert Haas
- Commit: Built-in JSON data type.
  (Tue Jan 31 11:48:23 2012 -0500)
  - Appears in PostgreSQL 9.2
  - Released September 2012
- No internal structure
- No indexing

A Summer of Code

- Google Summer of Code project, 2010
- Introduced initial JSON support in PostgreSQL
- authored by Joseph Adams
  - Mentored by Andrew Dunstan
  - Committed by Robert Haas
- Commit: Built-in JSON data type.
  (Tue Jan 31 11:48:23 2012 -0500)
  - Appears in PostgreSQL 9.2
  - Released September 2012
- No internal structure
- No indexing

A Boost of Unstructure

- Teodor Sigaev posts *nested hstore* support

  pgsql-hackers: nested hstore patch
  (Tue, 12 Nov 2013 22:35:31 +0400)

- JSONB comes to life

  Commit: Introduce jsonb, a structured format for storing json.
  (Sun Mar 23 16:40:19 2014 -0400)

- Binary storage
- Allows indexing

A Boost of Unstructure

- Teodor Sigaev posts *nested hstore* support

  pgsql-hackers: nested hstore patch
  (Tue, 12 Nov 2013 22:35:31 +0400)

- JSONB comes to life

  Commit: Introduce jsonb, a structured format for storing json.
  (Sun Mar 23 16:40:19 2014 -0400)

- Binary storage
- Allows indexing

A Surprise of Easterners

- February 2017: Oleg Bartunov posts about supporting the SQL standard syntax

  pgsql-hackers: SQL/JSON in PostgreSQL

  (Tue, 28 Feb 2017 22:08:43 +0300)

- A 431 kB, 15000 lines patch!
- Authors: Nikita Glukhov, Teodor Sigaev, Oleg Bartunov, Alexander Korotkov

A List of Functions

4.46.4 SQL/JSON functions

All manipulation (*e.g.*, retrieval, creation, testing) of SQL/JSON items is performed through a number of SQL/JSON functions.

There are nine such functions, categorized as SQL/JSON retrieval functions and SQL/JSON construction functions. The SQL/JSON retrieval functions are characterized by operating on JSON data and returning an SQL value (possibly a Boolean value) or a JSON value. The SQL/JSON construction functions return JSON data created from operations on SQL data or other JSON data.

The SQL/JSON retrieval functions are:

— <JSON value function>: extracts an SQL value of a predefined type from a JSON text.

— <JSON query>: extracts a JSON text from a JSON text.

— <JSON table>: converts a JSON text to an SQL table.

— <JSON predicate>: tests whether a string value is or is not properly formed JSON text.

— <JSON exists predicate>: tests whether an SQL/JSON path expression returns any SQL/JSON items.

The SQL/JSON construction functions are:

— <JSON object constructor>: generates a string that is a serialization of an SQL/JSON object.

— <JSON array constructor>: generates a string that is a serialization of an SQL/JSON array.

— <JSON object aggregate constructor>: generates, from an aggregation of SQL data, a string that is a serialization of an SQL/JSON object.

— <JSON array aggregate constructor>: generates, from an aggregation of SQL data, a string that is a serialization of an SQL/JSON array.

A *JSON-returning function* is an SQL/JSON construction function or JSON_QUERY.

A List of Functions (1)

The SQL/JSON retrieval functions are:

ISO/IEC 9075-2:2016(E)
4.46 JSON data handling in SQL

— <JSON value function>: extracts an SQL value of a predefined type from a JSON text.

— <JSON query>: extracts a JSON text from a JSON text.

— <JSON table>: converts a JSON text to an SQL table.

— <JSON predicate>: tests whether a string value is or is not properly formed JSON text.

— <JSON exists predicate>: tests whether an SQL/JSON path expression returns any SQL/JSON items.

A Definition of Datatype

> **3.1.6.29 JSON text**
> sequence of JSON tokens, which must be encoded in Unicode [Unicode] (UTF-8 by default); insignificant white space may be used anywhere in JSON text except within strings (where all white space is significant), numbers, and literals

A List of Functions (2)

The SQL/JSON construction functions are:

— <JSON object constructor>: generates a string that is a serialization of an SQL/JSON object.

— <JSON array constructor>: generates a string that is a serialization of an SQL/JSON array.

— <JSON object aggregate constructor>: generates, from an aggregation of SQL data, a string that is a serialization of an SQL/JSON object.

— <JSON array aggregate constructor>: generates, from an aggregation of SQL data, a string that is a serialization of an SQL/JSON array.

A *JSON-returning function* is an SQL/JSON construction function or JSON_QUERY.

A Disaster of Commitments

- First commit: March 22 2022, 17:32
- Immediately reverted at 19:56
- Further 9 commits: from March 27th 2022 until April 7th
- Plus various later fixups
- Everything (23 commits) reverted again in September 2022
- Fundamental problem: catching parse errors

A Disaster of Commitments

- First commit: March 22 2022, 17:32
- Immediately reverted at 19:56
- Further 9 commits: from March 27th 2022 until April 7th
- Plus various later fixups
- Everything (23 commits) reverted again in September 2022
- Fundamental problem: catching parse errors

A Disaster of Commitments

- First commit: March 22 2022, 17:32
- Immediately reverted at 19:56
- Further 9 commits: from March 27th 2022 until April 7th
- Plus various later fixups
- Everything (23 commits) reverted again in September 2022
- Fundamental problem: catching parse errors

A Disaster of Commitments

- First commit: March 22 2022, 17:32
- Immediately reverted at 19:56
- Further 9 commits: from March 27th 2022 until April 7th
- Plus various later fixups
- Everything (23 commits) reverted again in September 2022
- Fundamental problem: catching parse errors

A Capture of Errors

- *Soft errors* came about
- A new mechanism to capture errors before they're thrown
- More robust and performant

A Capture of Errors

- *Soft errors* came about
- A new mechanism to capture errors before they're thrown
- More robust and performant

```
277 /*
278  * "ereturn(context, dummy_value, ...);" is exactly the same as
279  * "errsave(context, ...); return dummy_value;".  This saves a bit
280  * of typing in the common case where a function has no cleanup
281  * actions to take after reporting a soft error.  "dummy_value"
282  * can be empty if the function returns void.
283  */
284 #define ereturn_domain(context, dummy_value, domain, ...)   \
285     do { \
286         errsave_domain(context, domain, __VA_ARGS__); \
287         return dummy_value; \
288     } while(0)
289
290 #define ereturn(context, dummy_value, ...)  \
291     ereturn_domain(context, dummy_value, TEXTDOMAIN, __VA_ARGS__)
```

A Capture of Errors (1)

```
242 /*----------
243  * Support for reporting "soft" errors that don't require a full transaction
244  * abort to clean up.  This is to be used in this way:
245  *      errsave(context,
246  *              errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
247  *              errmsg("invalid input syntax for type %s: \"%s\"",
248  *                     "boolean", in_str),
249  *              ... other errxxx() fields as needed ...);
250  *
251  * "context" is a node pointer or NULL, and the remaining auxiliary calls
252  * provide the same error details as in ereport().  If context is not a
253  * pointer to an ErrorSaveContext node, then errsave(context, ...)
254  * behaves identically to ereport(ERROR, ...).  If context is a pointer
255  * to an ErrorSaveContext node, then the information provided by the
256  * auxiliary calls is stored in the context node and control returns
257  * normally.  The caller of errsave() must then do any required cleanup
258  * and return control back to its caller.  That caller must check the
259  * ErrorSaveContext node to see whether an error occurred before
260  * it can trust the function's result to be meaningful.
261  *
262  * errsave_domain() allows a message domain to be specified; it is
263  * precisely analogous to ereport_domain().
264  *----------
265  */
266 #define errsave_domain(context, domain, ...)    \
267     do { \
268         struct Node *context_ = (context); \
269         pg_prevent_errno_in_scope(); \
270         if (errsave_start(context_, domain)) \
271             __VA_ARGS__, errsave_finish(context_, __FILE__, __LINE__, __func__); \
272     } while(0)
```

A Set of Updates

- Amit Langote picks up the baton

  pgsql-hackers: SQL/JSON revisited
  (Wed, 28 Dec 2022 16:28:29 +0900)

Attachment	Content-Type	Size
v1-0007-JSON_TABLE.patch	application/octet-stream	99.1 KB
v1-0010-Claim-SQL-standard-compliance-for-SQL-JSON-featur.patch	application/octet-stream	2.8 KB
v1-0009-Documentation-for-SQL-JSON-features.patch	application/octet-stream	47.7 KB
v1-0008-PLAN-clauses-for-JSON_TABLE.patch	application/octet-stream	70.7 KB
v1-0006-RETURNING-clause-for-JSON-and-JSON_SCALAR.patch	application/octet-stream	10.2 KB
v1-0005-SQL-JSON-functions.patch	application/octet-stream	52.0 KB
v1-0004-SQL-JSON-query-functions.patch	application/octet-stream	177.4 KB
v1-0001-Common-SQL-JSON-clauses.patch	application/octet-stream	24.9 KB
v1-0003-IS-JSON-predicate.patch	application/octet-stream	43.6 KB
v1-0002-SQL-JSON-constructors.patch	application/octet-stream	147.4 KB

A First of Many

- Álvaro Herrera commits SQL/JSON constructors
- It finally sticks!

  Commit: SQL/JSON: add standard JSON constructor functions
  (Wed Mar 29 12:11:36 2023 +0200)

- Squash of Amit's 0001, 0002, and parts of 0009

- `JSON_ARRAY()`
- `JSON_OBJECT()`
- `JSON_ARRAYAGG()`
- `JSON_OBJECTAGG()`

JSON_ARRAY()

```
JSON_ARRAY ( [ { value_expression [ FORMAT JSON ] } [, ...] ]
             [ { NULL | ABSENT } ON NULL ]
             [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ] ])

JSON_ARRAY ( [ query_expression ]
             [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ] ])
```

JSON_ARRAY()

```
JSON_ARRAY ( [ { value_expression [ FORMAT JSON ] } [, ...] ]
             [ { NULL | ABSENT } ON NULL ]
             [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ] ])

JSON_ARRAY ( [ query_expression ]
             [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ] ])
```

```
SELECT JSON_ARRAY(1+1, current_date,
                  '{"eek" : "ugly" }',
                  '{"happiness":"yes"}' FORMAT JSON);

                                json_array
-----------------------------------------------------------------------------
 [2, "2024-05-26", "{\"eek\" : \"ugly\" }", {"happiness":"yes"}]
```

JSON_ARRAY()

```
JSON_ARRAY ( [ { value_expression [ FORMAT JSON ] } [, ...] ]
             [ { NULL | ABSENT } ON NULL ]
             [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ] ])

JSON_ARRAY ( [ query_expression ]
             [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ] ])
```

```
SELECT JSON_ARRAY( SELECT relname FROM pg_class LIMIT 3 );
```

```
                                json_array
-----------------------------------------------------------------------
 ["postgres_log", "pg_toast_43619", "pg_toast_43619_index"]
```

JSON_ARRAY()

```
JSON_ARRAY ( [ { value_expression [ FORMAT JSON ] } [, ...] ]
             [ { NULL | ABSENT } ON NULL ]
             [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ] ])

JSON_ARRAY ( [ query_expression ]
             [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ] ])
```

```
SELECT (JSON_ARRAY( '{"answer" : 42}' FORMAT JSON,
                    '{"question": "?"}' FORMAT JSON
        RETURNING JSONB))[0];

   json_array
________________
 {"answer": 42}
```

JSON_ARRAY()

```
JSON_ARRAY ( [ { value_expression [ FORMAT JSON ] } [, ...] ]
             [ { NULL | ABSENT } ON NULL ]
             [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ] ] )

JSON_ARRAY ( [ query_expression ]
             [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ] ] )
```

```
SELECT JSON_ARRAY( 1, NULL, 3 NULL ON NULL );
```

```
  json_array
---------------
 [1, null, 3]
```

JSON_OBJECT()

```
JSON_OBJECT ( [ { key_expression { VALUE | ':' } value_expression
                  [ FORMAT JSON [ ENCODING UTF8 ] ] }[, ...] ]
                [ { NULL | ABSENT } ON NULL ]
                [ { WITH | WITHOUT } UNIQUE [ KEYS ] ]
               [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ] ])
```

JSON_OBJECT()

```
JSON_OBJECT ( [ { key_expression { VALUE | ':' } value_expression
                  [ FORMAT JSON [ ENCODING UTF8 ] ] }[, ...] ]
                [ { NULL | ABSENT } ON NULL ]
                [ { WITH | WITHOUT } UNIQUE [ KEYS ] ]
              [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ] ])
```

```
SELECT JSON_OBJECT('theKey' : 'theValue',
                   ('nother' || 'key') VALUE current_date,
                   'theKey' : null
                   ABSENT ON NULL);
```

```
                    json_object
-----------------------------------------------------------
 {"theKey" : "theValue", "notherkey" : "2024-05-27"}
```

JSON_OBJECT()

```
JSON_OBJECT ( [ { key_expression { VALUE | ':' } value_expression
                  [ FORMAT JSON [ ENCODING UTF8 ] ] }[, ...] ]
                [ { NULL | ABSENT } ON NULL ]
                [ { WITH | WITHOUT } UNIQUE [ KEYS ] ]
               [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ] ])
```

```
SELECT JSON_OBJECT('theKey' : 'value 1',
                   'theKey' : 'value 2'
                   WITH UNIQUE KEYS);
ERROR:  duplicate JSON object key value: "theKey"
```

JSON_OBJECT()

```
JSON_OBJECT ( [ { key_expression { VALUE | ':' } value_expression
                  [ FORMAT JSON [ ENCODING UTF8 ] ] }[, ...] ]
                [ { NULL | ABSENT } ON NULL ]
                [ { WITH | WITHOUT } UNIQUE [ KEYS ] ]
               [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ] ])
```

- Alert of consistency loss

```
SELECT JSON_OBJECT('the' || 'Key' : 'theValue',
                   ('the' || 'Key') VALUE current_date
                   WITHOUT UNIQUE KEYS
                   RETURNING jsonb);
       json_object
-------------------------------
 {"theKey": "2024-05-27"}
```

JSON_ARRAYAGG()

```
JSON_ARRAYAGG ( [ value_expression ] [ ORDER BY sort_expression ]
                [ { NULL | ABSENT } ON NULL ]
                [ RETURNING data_type
                          [ FORMAT JSON [ ENCODING UTF8 ] ] ])
```

JSON_ARRAYAGG()

```
JSON_ARRAYAGG ( [ value_expression ] [ ORDER BY sort_expression ]
                [ { NULL | ABSENT } ON NULL ]
                [ RETURNING data_type
                        [ FORMAT JSON [ ENCODING UTF8 ] ] ])
```

```
SELECT last_name, JSON_ARRAYAGG(first_name ORDER BY first_name)
       FROM actor GROUP BY last_name LIMIT 3;
```

last_name	json_arrayagg
AKROYD	["CHRISTIAN", "DEBBIE", "KIRSTEN"]
ALLEN	["CUBA", "KIM", "MERYL"]
ASTAIRE	["ANGELINA"]

JSON_OBJECTAGG()

```
JSON_OBJECTAGG ( [ { key_expression { VALUE | ':' } value_expression } ]
                 [ { NULL | ABSENT } ON NULL ]
                 [ { WITH | WITHOUT } UNIQUE [ KEYS ] ]
               [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ] ])
```

JSON_OBJECTAGG()

```
SELECT a.address, jsonb_pretty(
    JSON_OBJECTAGG(staff_id : format('%s %s', first_name, last_name)
                   RETURNING jsonb))
    FROM staff JOIN store USING (store_id) JOIN address a
                  ON (store.address_id = a.address_id)
GROUP BY store_id, a.address_id LIMIT 1;
```

address	jsonb_pretty
569 Baicheng Lane	{ "6": "Theo Harber", "27": "Mercedes Gislason", "36": "Flor Toy", "1421": "Drew Fahey", "1436": "Tyree Dicki", "1471": "Dan Kling" }

A Last of Sixteen

- Álvaro Herrera commits the `IS JSON` predicate

  Commit: SQL/JSON: support the IS JSON predicate
  (Fri Mar 31 22:34:04 2023 +0200)

- `IS JSON [VALUE]`
- `IS JSON ARRAY`
- `IS JSON OBJECT`
- `IS JSON SCALAR`

- Nothing else would get done for 16, sadly

A Last of Sixteen

- Álvaro Herrera commits the `IS JSON` predicate

  Commit: SQL/JSON: support the IS JSON predicate
  (Fri Mar 31 22:34:04 2023 +0200)

- `IS JSON [VALUE]`
- `IS JSON ARRAY`
- `IS JSON OBJECT`
- `IS JSON SCALAR`

- Nothing else would get done for 16, sadly

A Last of Sixteen

- Álvaro Herrera commits the `IS JSON` predicate
  Commit: SQL/JSON: support the IS JSON predicate
  (Fri Mar 31 22:34:04 2023 +0200)

- `IS JSON [VALUE]`
- `IS JSON ARRAY`
- `IS JSON OBJECT`
- `IS JSON SCALAR`

- Nothing else would get done for 16, sadly

```
SELECT '{"xyzxz": 456}' IS JSON OBJECT;
 ?column?
----------
 t
```

A Completion of Efforts

- Amit Langote runs the final marathon

  pgsql-hackers: remaining sql/json patches
  (Mon, 19 Jun 2023 17:31:57 +0900)

Attachment	Content-Type	Size
v1-0001-SQL-JSON-functions.patch	application/octet-stream	57.8 KB
v1-0004-Claim-SQL-standard-compliance-for-SQL-JSON-featur.patch	application/octet-stream	2.3 KB
v1-0003-JSON_TABLE.patch	application/octet-stream	160.3 KB
v1-0002-SQL-JSON-query-functions.patch	application/octet-stream	208.8 KB

An Update of Standards

4.8.3 Operations involving JSON values

<JSON parse> is an operator that parses a character string or binary string; i.e., converts JSON text to an SQL/JSON item, according to the rules of RFC 8259.

<JSON serialize> is an operator that, given a SQL/JSON value *JV*, returns either a character string or binary string value *SV* such that the result of a <JSON parse> with *SV* as input would be equivalent to *JV*.

<JSON object constructor> is an operator that, given zero or more name-value pairs, returns an SQL/JSON item that is an SQL/JSON object.

<JSON array constructor> is an operator that, given a list of values, returns an SQL/JSON item that is an SQL/JSON array.

<JSON scalar> is an operator that, given a value *V* of character string data type, numeric data type, Boolean data type, or datetime data type, returns an SQL/JSON item that is a SQL/JSON scalar whose value is *V*.

<JSON object aggregate constructor> is an operator that returns an SQL/JSON item that is an SQL/JSON object from a collection of rows.

<JSON array aggregate constructor> is an operator that returns an SQL/JSON item that is an SQL/JSON array from a collection of rows.

<JSON query> is an operator that, given an SQL/JSON item and an SQL/JSON path expression, returns an SQL/JSON item.

<JSON value function> is an operator that, given an SQL/JSON item and an SQL/JSON path expression, returns an SQL value of character string data type, numeric data type, Boolean data type, or datetime data type.

<JSON simplified accessor> provides a subset of the functionality of <JSON query> and <JSON value function> using a more convenient syntax.

<JSON table> is a kind of <derived table>, which may be used to query an SQL/JSON value as a table.

<JSON predicate> is a predicate that determines whether a given character string or JSON value satisfies the rules of RFC 8259 for JSON objects, JSON arrays, and/or JSON scalars.

<JSON exists predicate> is a predicates that evaluates an SQL/JSON path expression and determines if the result is an SQL null value, an empty SQL/JSON sequence, or a non-empty SQL/JSON sequence.

<JSON object constructor>, <JSON array constructor>, <JSON object aggregate constructor>, <JSON array aggregate constructor>, and <JSON query> have the option of returning a character string value or a binary string value instead of a JSON value, in which case the result is implicitly serialized.

A Companionship of Constructors

- Amit Langote commits more constructors

  Commit: Add more SQL/JSON constructor functions
  (Wed Jul 26 17:08:33 2023 +0900)

- `JSON()`
- `JSON_SCALAR()`
- `JSON_SERIALIZE()`

A Companionship of Constructors

- Amit Langote commits more constructors

  Commit: Add more SQL/JSON constructor functions
  (Wed Jul 26 17:08:33 2023 +0900)

- `JSON()`
- `JSON_SCALAR()`
- `JSON_SERIALIZE()`

```
JSON ( expression [ FORMAT JSON [ ENCODING UTF8 ]]
       [ { WITH | WITHOUT } UNIQUE [ KEYS ]] )
```

```
SELECT JSON('{"hike": [1, 2, {"answer":42, "question":null}]}');
```

```
                        json
-----------------------------------------------------
 {"hike": [1, 2, {"answer":42, "question":null}]}
```

A Companionship of Constructors

- Amit Langote commits more constructors

  Commit: Add more SQL/JSON constructor functions
  (Wed Jul 26 17:08:33 2023 +0900)

- `JSON()`
- `JSON_SCALAR()`
- `JSON_SERIALIZE()`

```
JSON_SCALAR ( expression )
```

```
SELECT JSON_SCALAR('xyzxz');
 json_scalar
-------------
 "xyzxz"
```

A Companionship of Constructors

- Amit Langote commits more constructors
  Commit: Add more SQL/JSON constructor functions
  (Wed Jul 26 17:08:33 2023 +0900)
- `JSON()`
- `JSON_SCALAR()`
- `JSON_SERIALIZE()`

```
JSON_SERIALIZE ( expression [ FORMAT JSON [ ENCODING UTF8 ] ]
                 [ RETURNING data_type
                       [ FORMAT JSON [ ENCODING UTF8 ] ] ] )
```

```
SELECT pg_typeof(JSON_SERIALIZE('{"whatever": "some values"}'
                 RETURNING text));
 pg_typeof
-----------
 text
```

A Definition of Paths

- Amit Langote commits SQL/JSON query functions

  Commit: Add SQL/JSON query functions
  (Thu Mar 21 17:07:03 2024 +0900)

- `JSON_QUERY()`
- `JSON_EXISTS()`
- `JSON_VALUE()`

- (Data from *pagila* sample database)

```
create table allstores as
with store_staff as (
select
   store_id,
   json_arrayagg(
         json_object('id' : staff.staff_id,
                     'name' : staff.first_name || ' ' ||
                              staff.last_name,
                'address' : json_object(
                              'street' : sa.address,
                            'district' : sa.district,
                                 'zip' : sa.postal_code
                                       )
                    )
                ) as staff
  from staff join store using (store_id)
             join address sa on (staff.address_id = sa.address_id)
  group by store_id)
  select json_arrayagg(json_object(
           'id' : store.store_id,
      'manager' : mgr.first_name || ' ' || mgr.last_name,
        'staff' : store_staff.staff
      'address' : json_object('street': stoaddr.address,
                              'district': stoaddr.district,
                                   'zip': stoaddr.postal_code),
                  returning jsonb)
                     ) as alldata
from store
  join staff mgr on (store.manager_staff_id = mgr.staff_id)
  join store_staff on (store.store_id = store_staff.store_id)
  join address stoaddr on (store.address_id = stoaddr.address_id);
```

```
{
  "staff": [
    {
      "id": 1479,
      "name": "Nicky Gottlieb",
      "address": {
        "zip": "53829",
        "addr1": "1819 Alessandria Loop",
        "addr2": "",
        "district": "Campeche"
      }
    },
    {
      "id": 1430,
      "name": "Isaias Wehner",
      "address": {
        "zip": "37815",
        "addr1": "1351 Sousse Lane",
        "addr2": "",
        "district": "Coahuila de Zaragoza"
      }
    }
  ],
  "address": {
    "zip": "53446",
    "address": "247 Jining Parkway",
    "district": "Banjul"
  },
  "manager": "Eddie Bartell"
```

JSON_QUERY()

```
JSON_QUERY ( context_item,
       path_expression [ PASSING { value AS varname } [, ...]]
            [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ] ]
      [ { WITHOUT | WITH [ CONDITIONAL | UNCONDITIONAL ] }
                                                [ ARRAY ] WRAPPER ]
      [ { KEEP | OMIT } QUOTES [ ON SCALAR STRING ] ]
      [ { ERROR | NULL | EMPTY [ ARRAY | OBJECT ] |
                                    DEFAULT expression } ON EMPTY ]
      [ { ERROR | NULL | EMPTY [ ARRAY | OBJECT ] |
                                    DEFAULT expression } ON ERROR ])
```

JSON_QUERY()

```
JSON_QUERY ( context_item,
       path_expression [ PASSING { value AS varname } [, ...]]
            [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ] ]
      [ { WITHOUT | WITH [ CONDITIONAL | UNCONDITIONAL ] }
                                          [ ARRAY ] WRAPPER ]
      [ { KEEP | OMIT } QUOTES [ ON SCALAR STRING ] ]
      [ { ERROR | NULL | EMPTY [ ARRAY | OBJECT ] |
                                    DEFAULT expression } ON EMPTY ]
      [ { ERROR | NULL | EMPTY [ ARRAY | OBJECT ] |
                                    DEFAULT expression } ON ERROR ])


SELECT json_query(alldata,
            '$[$storeidx].manager' PASSING 42 AS storeidx
                  RETURNING JSON)    FROM allstores;

        json_query
 ------------------
  "Kristel Bins"
```

JSON_QUERY()

```
select jsonb_pretty(json_query(alldata, '$[2 to 4].staff[*].name'
            returning jsonb with wrapper))
    from allstores;

     jsonb_pretty
_________________________
 [                       ↩
     "Mira Reynolds",    ↩
     "Lexie Von",        ↩
     "Jeanene Rippin",   ↩
     "Jeannie Cronin",   ↩
     "Moon Ondricka",    ↩
     "Juliette Kulas",   ↩
     "Thersa Daniel",    ↩
     "Dee Zboncak"       ↩
 ]
```

A Language of Inspection

- `jsonpath` documentation
  - Authored by Nikita Glukhov and Teodor Sigaev
  - Committed by Alexander Korotkov in PostgreSQL 12
    Commit: Partial implementation of SQL/JSON path language
    (Sat Mar 16 12:16:48 2019 +0300)

`$`	The "context item"
`$var`	A variable in the PASSING clause
`.key`	Object member accessor
`.*`	Accessor for all direct members of object
`[index]`	Array member accessor

A Language of Inspection (2)

`[*]` Accessor for all array members

`[index1, index2, ...]` Scattered member accesor

`[start_index to end_index]` Multi-item array member accessor

`.**` Recursive accessor for object

`.**{level}` Recursive accessor for object at specific level

`.**{start_level to end_level}` Recursive, given levels

`.function()` Function invocation

modes *lax* and *strict*

A Language of Inspection (2)

`[*]` Accessor for all array members

`[index1, index2, ...]` Scattered member accesor

`[start_index to end_index]` Multi-item array member accessor

`.**` Recursive accessor for object

`.**{level}` Recursive accessor for object at specific level

`.**{start_level to end_level}` Recursive, given levels

`.`*`function`*`()` Function invocation

modes *lax* and *strict*

A Filtering of Results

```
SELECT jsonb_pretty(json_query(alldata,
       '$ ? (@.staff[*].address.district == "Santiago")'
    RETURNING JSONB WITH ARRAY WRAPPER))
    FROM allstores;
```

```
                    jsonb_pretty
-----------------------------------------------------------
 [                                                        ↵
     {                                                    ↵
         "id": 243,                                       ↵
         "staff": [                                       ↵
             {                                            ↵
                 "id": 646,                               ↵
                 "name": "Joanie Schroeder",              ↵
                 "address": {                             ↵
                     "zip": "69517",                      ↵
                     "street": "532 Toulon Street",       ↵
                     "district": "Santiago"               ↵
                 }                                        ↵
             },                                           ↵
```

An Insistence of Filters

```
SELECT json_query(alldata,
'strict $[*] ? (@.staff[*].address.district == $loc).staff.size()'
  PASSING 'Santiago' AS loc ERROR ON ERROR)
FROM allstores;

 json_query
------------
 5

SELECT json_query(alldata,
  '$ ? (@.staff.size() > 7)
       ?  (@.address.zip like_regex "^55").manager'
  RETURNING TEXT OMIT QUOTES) FROM allstores;

   json_query
----------------
 Danial Quitzon
```

An Insistence of Filters

```
SELECT json_query(alldata,
'strict $[*] ? (@.staff[*].address.district == $loc).staff.size()'
  PASSING 'Santiago' AS loc ERROR ON ERROR)
FROM allstores;

 json_query
------------
 5

SELECT json_query(alldata,
  '$ ? (@.staff.size() > 7)
       ?  (@.address.zip like_regex "^55").manager'
  RETURNING TEXT OMIT QUOTES) FROM allstores;

   json_query
----------------
 Danial Quitzon
```

JSON_VALUE()

```
JSON_VALUE ( context_item,
       path_expression [ PASSING { value AS varname } [, ...]]
       [ RETURNING data_type ]
       [ { ERROR | NULL | DEFAULT expression } ON EMPTY ]
       [ { ERROR | NULL | DEFAULT expression } ON ERROR ])
```

```
SELECT JSON_VALUE(alldata,
'strict $[*] ? (@.staff[*].address.district == $loc).staff[1].name'
  PASSING 'Santiago' AS loc ERROR ON ERROR)
FROM allstores;

 json_value
------------
 Love Feest
```

JSON_VALUE()

```
  JSON_VALUE ( context_item,
         path_expression [ PASSING { value AS varname } [, ...]]
         [ RETURNING data_type ]
         [ { ERROR | NULL | DEFAULT expression } ON EMPTY ]
         [ { ERROR | NULL | DEFAULT expression } ON ERROR ])
```

```
SELECT JSON_VALUE(alldata,
'strict $[*] ? (@.staff[*].address.district == $loc).staff[1].name'
  PASSING 'Santiago' AS loc ERROR ON ERROR)
FROM allstores;
```

```
 json_value
------------
 Love Feest
```

JSON_EXISTS()

```
JSON_EXISTS ( context_item,
              path_expression [ PASSING { value AS varname } [, ...]]
              [ { TRUE | FALSE | UNKNOWN | ERROR } ON ERROR ])
```

- Returns (Boolean) whether an item matches the search expression
- `ERROR ON ERROR` recommended

A full-circle of Models

- Amit Langote commits JSON_TABLE

  Commit: Add basic JSON_TABLE() functionality
  (Thu Apr 4 20:20:15 2024 +0900)

- Amit Langote adds NESTED clause to JSON_TABLE

  Commit: JSON_TABLE: Add support for NESTED paths and columns
  (Mon Apr 8 16:14:13 2024 +0900)

JSON_TABLE()

```
JSON_TABLE (
    context_item, path_expression [ AS json_path_name ]
    [ PASSING { value AS varname } [, ...] ]
    COLUMNS ( json_table_column [, ...] )
    [ { ERROR | EMPTY } ON ERROR ]
)
                          where json_table_column is:
name FOR ORDINALITY
| name type
      [ FORMAT JSON [ENCODING UTF8]]
      [ PATH path_expression ]
      [ { WITHOUT | WITH } [ CONDITIONAL | UNCONDITIONAL ] [ARRAY] WRAPPER ]
      [ { KEEP | OMIT } QUOTES [ ON SCALAR STRING ] ]
      [ { ERROR | NULL | EMPTY { ARRAY | OBJECT } | DEFAULT expression }
                                                          ON EMPTY ]
      [ { ERROR | NULL | EMPTY { ARRAY | OBJECT } | DEFAULT expression }
                                                          ON ERROR ]
| name type EXISTS [ PATH path_expression ]
      [ { ERROR | TRUE | FALSE | UNKNOWN } ON ERROR ]
| NESTED [ PATH ] path_expression [ AS json_path_name ]
                  COLUMNS ( json_table_column [, ...] )
```

JSON_TABLE()

```
SELECT j.store_id, j.manager, jsonb_pretty(address)
  FROM allstores,
         json_table(alldata,
                '$[$storeid]' PASSING 42 AS storeidx
                COLUMNS (
                      store_id  integer path  '($.id)',
                      manager   text path '($.manager)',
                      address   jsonb path '($.address)'
                )
        ) j;
```

store_id	manager	jsonb_pretty
46	Kristel Bins	{ "zip": "94352", "street": "1213 Ranchi Parkway", "district": "Karnataka" }

JSON_TABLE()

```
SELECT j.store_id, j.manager, jsonb_pretty(address)
  FROM allstores,
         json_table(alldata,
                '$[$storeid]' PASSING 42 AS storeidx
                COLUMNS (
                      store_id  integer path  '($.id)',
                      manager   text path '($.manager)',
                      address   jsonb path '($.address)'
                )
         ) j;
```

store_id	manager	jsonb_pretty
46	Kristel Bins	{ ⮒ "zip": "94352", ⮒ "street": "1213 Ranchi Parkway", ⮒ "district": "Karnataka" ⮒ }

JSON_TABLE()

```
SELECT j.*
  FROM allstores,
         json_table(alldata,
                'strict $[*] ? (@.id == $storeid)' PASSING 42 AS storeidx
                COLUMNS (
                      store_id  integer path  '($.id)',
                      manager   text path '($.manager)',
                      NESTED '$.staff[*]' columns
                        (name text path '$.name',
                         straddress text path '$.address.street'
                ))
        ) j;
```

```
-[ RECORD 1 ]----------------------
store_id    | 42
manager     | Lincoln Wisoky
name        | Norris Wilderman
straddress  | 1135 Izumisano Parkway
-[ RECORD 2 ]----------------------
store_id    | 42
manager     | Lincoln Wisoky
name        | Lester Stehr
straddress  | 698 Otsu Street
-[ RECORD 3 ]----------------------
store_id    | 42
manager     | Lincoln Wisoky
name        | Brendon Thiel
straddress  | 1966 Amroha Avenue
```

A Word of Wisdom

- JSON in PostgreSQL: how to use it right
- Laurenz Albe
- https://www.cybertec-postgresql.com/en/json-postgresql-how-to-use-it-right/

Thanks!

Questions?

Álvaro Herrera, EDB
alvherre@alvh.no-ip.org
alvaro.herrera@enterprisedb.com
Mastodon: https://lile.cl/@alvherre/