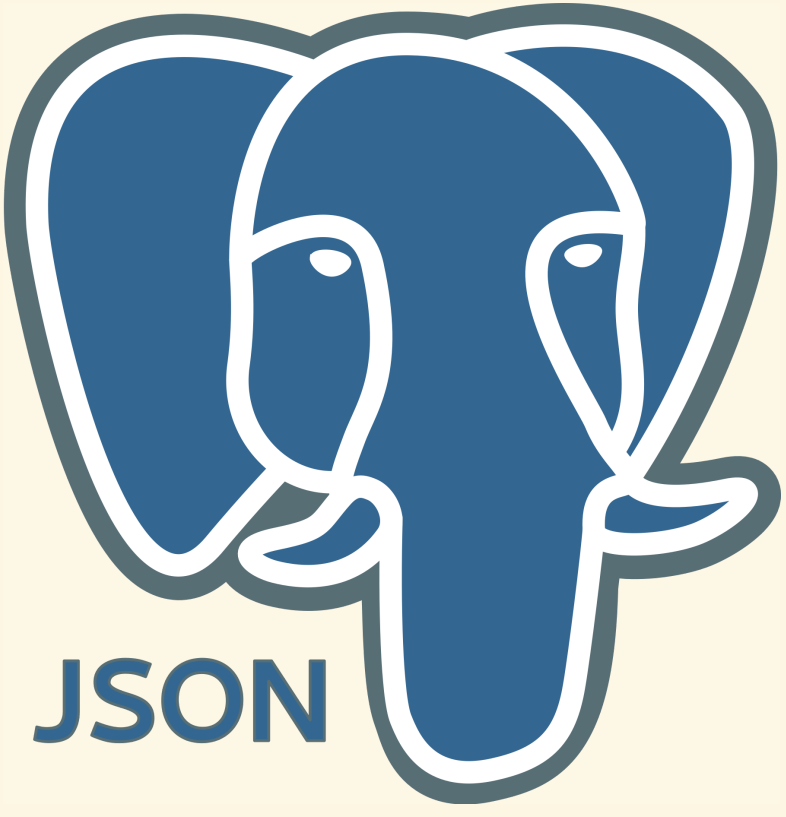


POSTGRE(NO)SQL

WHY USING JSON

Flexible schema (≠ destructured)

Open Library, Wikidata, …

Data denormalization

Avoid complex joins and reduce number of queries (e.g. EAV model)

Native JSON documents storage

JSON responses from REST APIs

THE OLD WAY

Before NoSQL and native JSON

EAV (ENTITY-ATTRIBUTE-VALUE) MODEL

One table contains the entities, another table contains the names of the properties (attributes) and a third table links the entities with their attributes and holds the value. This gives you the flexibility for having ***different sets of properties*** *[…].*

Selecting one or more entities based on 1 attribute value requires 2 ***joins*** *[…]. Also, the properties usually are all stored as strings, which results in* ***type casting****, both for the result as for the WHERE clause.*

JSONB has potential for greatly ***simplifying schema design without sacrificing query performance****.*

Replacing EAV with JSONB in PostgreSQL

via Wikipedia / Entity–attribute–value model

PLAIN TEXT AND CLIENT-SIDE MAPPING

Store JSON data in a text field and process it client-side.

- 👎 No SQL querying features
- 👎 Mapping overhead on client-side
- 👎 Multiple client bring duplicated code

THE MODERN WAY

2013 and beyond

2012-09-10

PostgreSQL 9.2 introduces `json` data type

- Store an exact copy of the input
  - Preserve order of keys
  - Preserve duplicate key/value pairs
- JSON validation (according to RFC-7159)
- JSON-specific functions and operators
  - Must *reparse input* on each execution
  - If duplicate key/value pairs are present, only the last value is considered

2 YEARS LATER (2014-12-18)

PostgreSQL 9.4 introduces `jsonb` data type

- Slower to input due to parsing overhead
  - Does not preserve the order of object keys
  - Does not keep duplicate object keys (in case of duplicate keys, only last value is kept)
- Faster to process (no need to reparse input)
- Support indexing
- Use PostgreSQL types to map JSON primitive types

1 YEAR LATER (2015)

3rd August: *MySQL 5.7.8* introduces JSON data type

14th October: *SQLite 3.9.0* introduces JSON data type

JSONB OPERATIONS

SELECT

UPDATE

QUERY

```
CREATE TABLE authors (
    type character varying,
    key character varying NOT NULL PRIMARY KEY,
    revision integer,
    last_modified timestamp without time zone,
    json jsonb
);

CREATE INDEX authors_idx_gin ON authors USING gin (json);
CREATE INDEX authors_idx_ginp ON authors USING gin (
    json jsonb_path_ops
);
```

```
{
    "key": "/authors/OL2623297A",
    "name": "Sir Arthur Conan Doyle",
    "photos": [5541405],
    "created": {
        "type": "/type/datetime", "value": "2008-04-29T13:35:46.876380"
    },
    "remote_ids": {
        "wikidata": "Q35610"
    },
    "last_modified": {
        "type": "/type/datetime", "value": "2017-03-31T14:21:52.424685"
    }
}
```

SELECT

SELECT BY ⌄ \ RESULT TYPE ›	jsonb	text
Object key/array index	->	->>
JSON path	#>	#>>

OBJECT KEY

```
SELECT json->'name', json->>'name', -- By object key
       json #> '{name}', json #>> '{name}' -- By JSON path
FROM authors;
```

```
SELECT json->'remote_ids'->'wikidata',
       json->'remote_ids'->>'wikidata',
       json #> '{remote_ids,wikidata}',
       json #>> '{remote_ids,wikidata}'
FROM authors;
```

If a field doesn't exist, NULL is returned.

ARRAY ELEMENT

```
-- Positive index (starting from 0)
SELECT json->'photos'->0, json->'photos'->>0, -- By index
       json #> '{photos, 0}', json #>> '{photos, 0}' -- By JSON path
FROM authors;
```

```
-- Negative indices count from the end of the array
SELECT json->'photos'->-1, json->'photos'->>-1, -- By index
       json #> '{photos, -1}', json #>> '{photos, -1}' -- By JSON path
FROM authors;
```

If an array element doesn't exist, NULL is returned.

UPDATE

Concatenate

Delete

`jsonb_strip_nulls`

`jsonb_set`

CONCATENATE (| |)

```
-- ARRAY
SELECT '["EUR", "USD"]'::jsonb || '"GBP"'::jsonb;
-- '["EUR", "USD", "GBP"]'
SELECT '["EUR", "USD"]'::jsonb || '{"GBP": "Pound Sterling"}'::jsonb;
-- '["EUR", "USD", {"GBP": "Pound Sterling"}]'
SELECT '["EUR", "USD"]'::jsonb || '["GBP", "CAD"]'::jsonb;
-- '["EUR", "USD", "GBP", "CAD"]'
```

```
-- OBJECTS
-- If duplicate keys are found, only last value is kept
SELECT '{"EUR": "Euro", "USD": "United States Dollar"}' ::jsonb ||
       '{"GBP": "Pound Sterling", "USD": "US Dollar"}' ::jsonb;
-- '{"EUR": "Euro", "GBP": "Pound Sterling", "USD": "US Dollar"}'
```

DELETE (- , #-)

```
-- OBJECT FIELD
SELECT '{"EUR": "Euro", "USD": "United States Dollar",
         "GBP": "Pound Sterling"}'::jsonb - 'USD', -- By key name
       '{"EUR": "Euro", "USD": "United States Dollar",
         "GBP": "Pound Sterling"}'::jsonb #- '{"USD"}'; -- By key path
-- '{"EUR": "Euro", "GBP": "Pound Sterling"}'
```

```
-- ARRAY ELEMENT
SELECT '["EUR", "USD", "GBP"]'::jsonb - 1, -- '["EUR", "GBP"]'
       '["EUR", "USD", "GBP"]'::jsonb #- '{1}', -- '["EUR", "GBP"]'
       '["EUR", "USD", "GBP"]'::jsonb - -1, -- '["EUR", "USD"]'
       '["EUR", "USD", "GBP"]'::jsonb #- '{-1}'; -- '["EUR", "USD"]'
```

JSONB_STRIP_NULLS(FROM_JSON)

```
SELECT jsonb_strip_nulls('{"EUR": "Euro", "GBP": null}'::jsonb);
-- '{"EUR": "Euro"}'
```

Remove object fields with `null` value.

```
SELECT jsonb_strip_nulls('["EUR", null, null, "CAD"]'::jsonb);
-- '["EUR", null, null, "CAD"]'
```

Has no effects on arrays with `null` elements.

JSONB_SET(TARGET, PATH, NEW_VALUE[, CREATE_IF_MISSING])

```
SELECT jsonb_set(
    '{"code": "EUR", "name": "Euro", "symbol": null}', '{symbol}', '"€"'
); -- '{"code": "EUR", "name": "Euro", "symbol": "€"}'
```

```
SELECT jsonb_set(
    '{"code": "EUR", "name": "Euro"}', '{symbol}', '"€"'
); -- '{"code": "EUR", "name": "Euro", "symbol": "€"}'
SELECT jsonb_set(
    '{"code": "EUR", "name": "Euro"}', '{symbol}', '"€"', false
); -- '{"code": "EUR", "name": "Euro"}'
```

Update/insert a JSON value at given object path and/or array position.

QUERY

Existence

Containment

Expansion

EXISTENCE (?, ? | , ?&)

```
-- ARRAY ELEMENTS
SELECT '["EUR", "USD", "GBP"]'::jsonb ? 'EUR'; -- true
SELECT '["EUR", "USD", "GBP"]'::jsonb ?| ARRAY['EUR', 'CAD']; -- true
SELECT '["EUR", "USD", "GBP"]'::jsonb ?& ARRAY['EUR', 'CAD']; -- false
-- Work only with elements on top-level array
SELECT '["EUR", ["USD", "GBP"]]'::jsonb ? 'USD'; -- false
```

```
-- OBJECT KEYS
SELECT '{"EUR": "Euro", "USD": "United States Dollar", "GBP":
         "Pound Sterling"}'::jsonb ? 'EUR'; -- true
SELECT '{"EUR": "Euro", "USD": "United States Dollar", "GBP":
         "Pound Sterling"}'::jsonb ?| ARRAY['EUR', 'CAD']; -- true
SELECT '{"EUR": "Euro", "USD": "United States Dollar", "GBP":
         "Pound Sterling"}'::jsonb ?& ARRAY['EUR', 'CAD']; -- false
-- Work only with keys on top-level object
SELECT '{"currencies": {"EUR": "Euro", "USD": "United States Dollar",
         "GBP": "Pound Sterling"}}'::jsonb ? 'EUR'; -- false
```

Check if a string appears as an object key or array element at the top level.

CONTAINMENT (@>, <@)

```
-- PRIMITIVE TYPES
-- Check absolute equality
SELECT '"EUR"'::jsonb @> '"EUR"'::jsonb; -- true
SELECT '"EUR"'::jsonb @> '"GBP"'::jsonb; -- false
```

```
-- ARRAYS
-- Order of elements is ignored
-- Duplicate elements are considered only once
SELECT '["EUR", "GBP", "USD"]'::jsonb
       @> '["GBP", "EUR", "EUR"]'::jsonb; -- true
SELECT '["EUR", "GBP", "USD"]'::jsonb @> '"EUR"'::jsonb; -- true
SELECT '["EUR", "GBP", "USD"]'::jsonb @> '"CAD"'::jsonb; -- false
```

```
-- OBJECTS
-- Structure and data contents must both match
SELECT '{"id": 12345, "name": "Nicola", "surname": "Moretto"}' ::jsonb
       @> '{"name": "Nicola"}'; -- true
SELECT '{"id": 12345, "name": "Nicola", "surname": "Moretto"}' ::jsonb
       @> '{"name": "Francesco"}'; -- false
```

EXPANSION (PART I: OBJECTS)

```
-- Return a record for each object field
SELECT jsonb_object_keys(
    '{"code": "EUR", "name": "Euro", "symbol": "€"}' ::jsonb
);
```

```
-- Return a record with 4 columns and values from matching object keys
SELECT *
FROM jsonb_to_record(
    '{"code":"EUR", "number":478, "name":"Euro", "symbol":"€"}' ::jsonb
) AS r(code text, number int, name text, symbol text);
```

EXPANSION (PART II: ARRAYS)

```
-- Return a record for each array element (as jsonb or text)
SELECT jsonb_array_elements('["EUR", "GBP", "CAD", "USD"]'::jsonb),
       jsonb_array_elements_text('["EUR", "GBP", "CAD", "USD"]'::jsonb);
```

```
-- Return for each object a record with two columns (code and name)
-- and values from matching object fields
SELECT *
FROM jsonb_to_recordset('[
    {"code": "GBP", "name": "Pound Sterling", "symbol": "£"},
    {"code": "EUR", "name": "Euro", "symbol": "€"},
    {"code": "USD", "name": "United States Dollar", "symbol": "$"}
]'::jsonb) AS c(code text, name text)
ORDER BY code ASC; -- Sort records by currency ISO code
```

JSONB INDEXING

```
CREATE INDEX ON authors USING GIN (?)
```

OPERATOR	TARGET	
	Table column	Index expression
jsonb_ops	`json`	`(json->'name')`
jsonb_path_ops	`json jsonb_path_ops`	`(json->'name') jsonb_path_ops`

INDEX EXPRESSIONS

Index expression is required if JSONB operator is not applied to table column

An index column need not be just a column of the underlying table, but can be a function or scalar expression computed from one or more columns of the table.

PostgreSQL 9.6 Documentation / Indexes / Indexes on Expressions

`JSONB_OPS` (DEFAULT)

Creates an index item for each **key** and value in the data.

- 👍 Support key-exists (?, ?& and ?|) and path/value-exists operators (@>, <@)
- 👎 Slower search than `jsonb_path_ops`
- 👎 Bigger index than `jsonb_path_ops` on same data

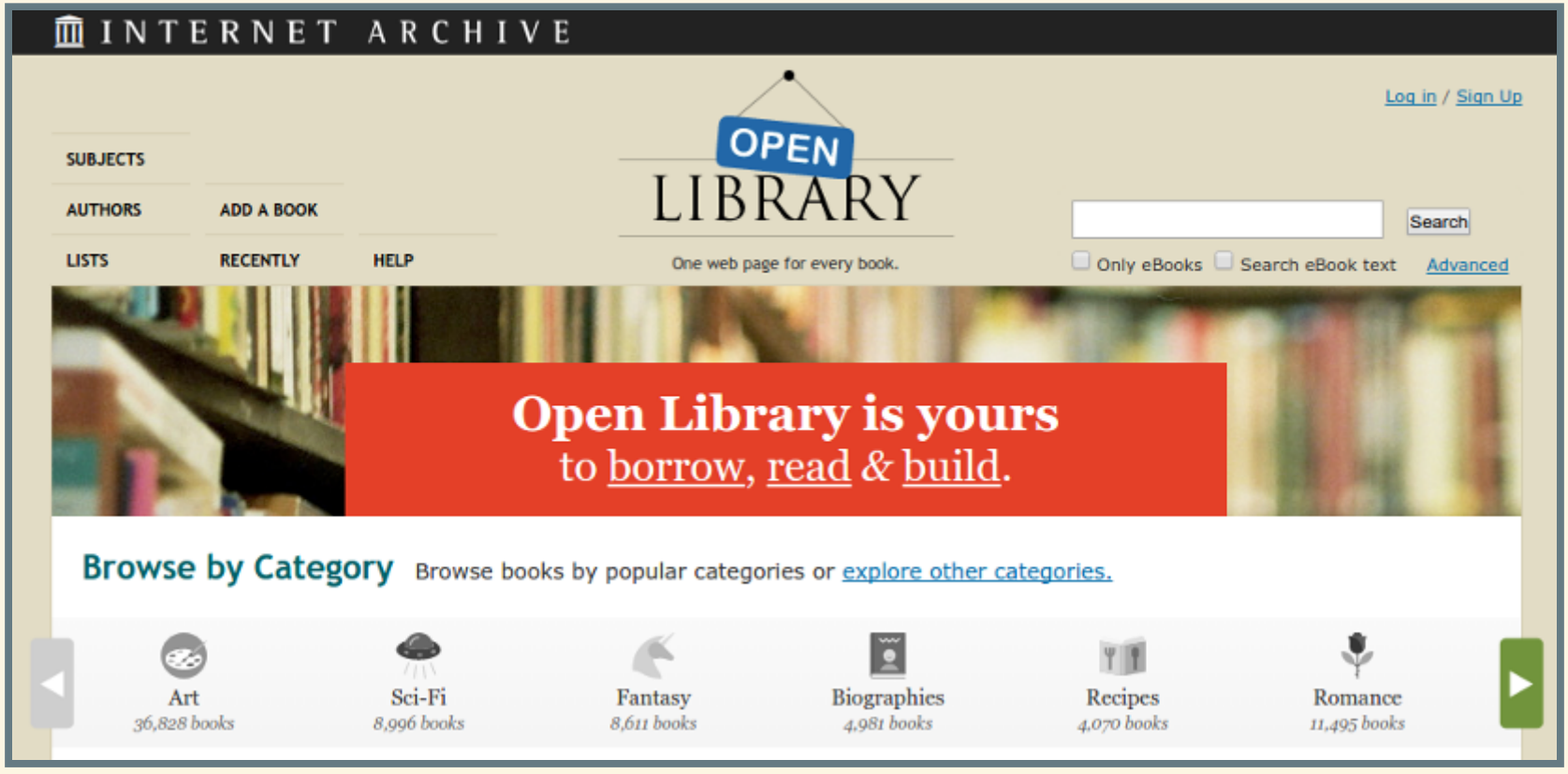
JSONB_PATH_OPS

Creates an index entry for each **value** and the key(s) leading to it.

- 👍 Best for containment queries
- 👍 Faster search than `jsonb_ops`
- 👍 Smaller index than `jsonb_ops` on same data
- 👎 Support only path/value-exists operators (@>, <@)
- 👎 No entries for JSON structures without value

DEMO

6.964.927 authors



SELECTING RECORD

```
SELECT * FROM authors
WHERE key = '/authors/OL2623297A';
-- Index Scan using PRIMARY KEY on authors
```

```
SELECT * FROM authors
WHERE json->>'key' = '/authors/OL2623297A';
-- Seq Scan on authors
```

```
SELECT * FROM authors
WHERE json @> '{"key": "/authors/OL2623297A"}'::jsonb;
-- Index Scan on authors_idx_ginp
```

```
SELECT * FROM authors
WHERE json @> '{"name": "Sir Arthur Conan Doyle"}'::jsonb;
-- Index Scan on authors_idx_ginp
```

COUNTING RECORDS

```
SELECT COUNT(*) FROM authors
WHERE json->'remote_ids' ? 'wikidata';
-- Seq Scan on authors
```

Existence operator can only use simple index for top-level keys, remember?

```
-- Create an index targeting the specifc key
CREATE INDEX authors_idx_gin_remoteids
ON authors USING GIN ((json -> 'remote_ids'));
```

```
SELECT COUNT(*) FROM authors
WHERE json->'remote_ids' ? 'wikidata';
-- Index Scan on authors_idx_gin_remoteids
```

AGGREGATING RECORDS

```
CREATE INDEX authors_idx_ginp_created
ON authors USING GIN ((json -> 'created') jsonb_path_ops);
```

```
-- Number of records added each day of last 2 months
SELECT (json #>> '{created,value}')::timestamp::date AS day,
       COUNT(*) AS records
FROM authors
WHERE json->'created' @> '{"type": "/type/datetime"}'
  AND (json #>> '{created,value}')::timestamp
      >= (CURRENT_DATE - INTERVAL '2 month')
GROUP BY 1
ORDER BY 1 DESC
-- Bitmap Index Scan on authors_idx_ginp_created
```

You can aggregate on an arbitrary date part using `date_trunc` function

```
CREATE MATERIALIZED VIEW mv_authors_cts AS
SELECT key, (json #>> '{created,value}')::timestamp AS ts
FROM authors
WHERE json->'created' @> '{"type": "/type/datetime"}'
WITH DATA;
CREATE INDEX mv_authors_cts_ts_idx ON mv_authors_cts(ts);

-- Remember to refresh materialized view if source table changes!
REFRESH MATERIALIZED VIEW mv_authors_cts;
```

```
SELECT ts::date AS day, COUNT(*) AS records
FROM mv_authors_cts
WHERE ts >= (CURRENT_DATE - INTERVAL '2 month')
GROUP BY 1
ORDER BY 1 DESC
-- Bitmap Index Scan on mv_authors_lmts_ts_idx
```

Thank you!

Full presentation and curated links on GitHub

 Nicola Moretto

 n.moretto@nicolamoretto.eu

 nicola88 (GitHub)

 nicola88 (BitBucket)