JSONB in PostgreSQL

CMPU4003 ADVANCED DATABASES

JSON vs JSONB

JSON:
- Textual
- Preserves whitespace/key order
- Duplicates remain (in text)

JSONB:
- Binary
- Canonical order
- Removes duplicates (last wins)

JSONB supports indexing and most operators
- Default choice

Why JSONB?

Offers flexibility for semi-structured data

- E.g. preferences, events, attributes

Provides rich operators and JSONPath

Allows use of GIN indexing for nested keys

- Prefer columns when shape is stable and heavily queried

Core operators you’ll use

```
-- Access
profile->'prefs'                       -- JSON
profile->>'name'                       -- text
profile#>'{addr,city}'                 -- JSON at path
profile#>>'{addr,city}'                -- text at path

-- Existence / containment
profile ? 'verified'                   -- key exists
profile @> '{"addr":{"country":"IE"}}'::JSONB

-- Arrays
JSONB_array_elements(profile#>'{prefs,langs}')  -- SRF

-- Merge & delete
profile || '{"verified": true}'::JSONB
profile - 'age'
profile #- '{addr,city}'
```

JSONPath

```
CREATE TABLE orders (
    id serial primary key,
    customer text,
    props JSONB
);

Suppose in props we have:

{
  "items": [
    { "name": "Apple", "qty": 3, "price": 0.5 },
    { "name": "Banana", "qty": 1, "price": 0.2 },
    { "name": "Orange", "qty": 5, "price": 0.4 }
  ]
}

-- Any item with quantity >=2
JSONB_path_exists(props, '$.items[*] ? (@.qty >= 2)’)

-- All item names
JSONB_path_query(props, '$.items[*].name’)


-- All prices
JSONB_path_query(props, '$.items[*].price')
```

Indexing strategies

Use GIN on JSONB for containment/exists filters

Generated columns + B-Tree for hot fields and sorting

- A *hot field* is simply a field in your JSON (or a regular column) that you **query or sort on very often**.

Use Partial indexes to keep size under control

- Sorting on JSON?
  - Use a generated column and index it.

Indexing examples

```
-- General GIN indexes
CREATE INDEX enrollments_grades_gin ON enrollments USING GIN
(grades);
CREATE INDEX students_profile_gin   ON students  USING GIN
(profile);

-- Promote a hot field
ALTER TABLE enrollments
  ADD COLUMN final_mark int GENERATED ALWAYS AS ((grades-
>>'final')::int) STORED;
CREATE INDEX enrollments_final_idx ON enrollments(final_mark);

-- Partial index example
CREATE INDEX enrollments_has_final ON enrollments USING GIN
(grades)
WHERE grades ? 'final';
```

Data quality and constraints

```
ALTER TABLE enrollments
  ADD CONSTRAINT grades_is_object
  CHECK (JSONB_typeof(grades) = 'object');

ALTER TABLE enrollments
  ADD CONSTRAINT final_0_100 CHECK (
    (grades ? 'final') IS NOT TRUE
    OR ((grades->>'final')::int BETWEEN 0 AND 100)
);
```

Modeling patterns

Hybrid:

- If you have stable IDs and frequently filtered fields as columns with a long-tail in JSONB

Event log:

- props JSONB per event; index name + common keys

Attribute bag: keep in JSONB; promote stable keys to columns later

Performance and Gotchas

Cast explicitly

- JSON numbers → ::int/::numeric

Missing vs null

- JSON null ≠ SQL NULL; use ? to test presence

Duplicate keys

- JSONB keeps only the last; avoid duplicates

GIN indexes can be large → consider partial indexes / generated columns

Use LATERAL carefully

- Explode arrays only when needed

Rules of Thumb

Start flexible with JSONB while requirements churn

As fields stabilize and become critical - promote to columns based on usage patterns

Index what you filter/sort on

Keep indexes lean