CMPU4003 Advanced Databases

Working with JSONB in PostgreSQL

**Goal:** Learn practical patterns for storing, querying, updating, and indexing semi‑structured academic data (students, subjects, grades, institutions) using jsonb in PostgreSQL.

# Setup

* If you have a PostgreSQL installation from last year this should be sufficient and you do not need to setup a new one.
* If you need to setup PostgreSQL there are a number of options:
  + Setup a Docker installation/Dev Container
    - Follow the instructions in Option 1 Docker from command line.txt
    - OR
    - Follow the instructions in Option 2 Docker in an IDE.txt
    - OR
    - Follow the instructions in Option 3 Dev Container in VS Code.txt
  + Use Supabase.com
    - Follow the instructions in Option 4 Supabase.txt
* Once you have PostgreSQL setup create a schema for this lab and set the search path so that Postgres will use this schema:

CREATE SCHEMA IF NOT EXISTS jsonb\_lab;  
SET search\_path = jsonb\_lab;

# Practical Tasks

Universities often record structured data (student IDs, subject codes) and semi‑structured data (metadata about assessments, remarks, evolving grading rubrics). jsonb helps store flexible information without altering schemas constantly. We are going to model students, enrollments and grades as JSON B.

## Create and populate the tables in the jsonb\_lab schema.

You are going to create two tables students and enrollments.

In students you will have a column Profiles which will store personal metadata (age, major, languages, sports, exchange) of type JSONB.

In enrollments you will have a column Grades which will store different assessment structures (simple scores, arrays of assignments, nested project details) of type JSONB.

# Drop and Create Tables

DROP TABLE IF EXISTS students CASCADE;  
CREATE TABLE students (  
 student\_id SERIAL PRIMARY KEY,  
 name TEXT NOT NULL,  
 university TEXT NOT NULL,  
 profile JSONB NOT NULL DEFAULT '{}'::jsonb  
);  
  
DROP TABLE IF EXISTS enrollments;  
CREATE TABLE enrollments (  
 enrollment\_id BIGSERIAL PRIMARY KEY,  
 student\_id INT REFERENCES students(student\_id),  
 subject\_code TEXT NOT NULL,  
 year INT NOT NULL,  
 semester INT NOT NULL,  
 grades JSONB NOT NULL -- stores assignments, exams, comments  
);

**-- Insert Sample Data**

INSERT INTO students (name, university, profile) VALUES  
('Alice Johnson', 'TU Dublin', '{"age":22,"major":"CS","languages":["en","fr"]}'),  
('Brian Smith', 'UCD', '{"age":24,"major":"Math","sports":["football"]}'),  
('Chloe Lee', 'Trinity', '{"age":21,"major":"Engineering","exchange":true}');  
  
INSERT INTO enrollments (student\_id, subject\_code, year, semester, grades) VALUES  
(1, 'DB4003', 2023, 1, '{"midterm":78,"final":85,"remarks":"Good progress"}'),  
(1, 'ML4001', 2023, 2, '{"assignments":[{"name":"A1","mark":40},{"name":"A2","mark":45}],"final":82}'),  
(2, 'DB4003', 2023, 1, '{"midterm":65,"final":70,"remarks":"Needs work"}'),  
(3, 'CS4090', 2023, 2, '{"project":{"title":"IoT","mark":88},"oral\_exam":90}');

Explanation:

Note that profile is flexible: Each student has a slightly different structure (some have languages, some sports, some exchange).

|  |  |  |
| --- | --- | --- |
| Alice Johnson  {  "age": 22,  "major": "CS",  "languages": ["en", "fr"]  }  Keys: age (number), major (string), languages (array of strings). | Brian Smith  {  "age": 24,  "major": "Math",  "sports": ["football"]  }  Keys: age, major, sports (array of strings). | Chloe Lee  {  "age": 21,  "major": "Engineering",  "exchange": true  }  Keys: age, major, exchange (boolean). |

## JSONB Basics

->> means **extract a JSON field as text**.

-> means **extract a JSON field as JSON**.

#> lets you navigate deeper using a **path array**.

the ? operator means: **“Does this JSON object contain the given key?”**

:: is the **type cast operator**.

Try these queries:

-- Extract fields  
SELECT name,  
 profile->>'major' AS major,  
 (profile->>'age')::int AS age  
FROM students;

*-- Extract fields as Json and text to illustrate the operator*

*--no different in visual editor will matter in applications*.

**SELECT** name,

**profile**->'major' AS major\_json,

**profile**->>'major' AS major\_text

**FROM** students;

-- Extract fields casting age to be type integer  
SELECT name,  
 profile->>'major' AS major,  
 (profile->>'age')::int AS age  
FROM students;

-- Does profile contain exchange info?  
SELECT name FROM students WHERE profile ? 'exchange';  
  
-- Subjects with final mark >= 80 where final is cast as an integer  
SELECT subject\_code, (grades->>'final')::int AS final  
FROM enrollments  
WHERE (grades->>'final')::int >= 80;

*-- Suppose we are looking for students with languages as part of their profile. We know Alice has a profile with {"languages":["en","fr"]}*

*-- profile#>'{languages}' → extracts the whole array as JSON.*

*-- profile#>>'{languages,0}' → navigates into the array (0 = first element) and returns text.*

**SELECT** name,

profile#>'{languages}' AS langs\_json,

profile#>>'{languages,0}' AS first\_lang

**FROM** students

**WHERE** profile ? 'languages';

## 2.3 Working with Arrays and Nested Objects

-- Expand assignments into rows to get the grades for each assignment for subject ML4001  
SELECT e.subject\_code, a->>'name' AS assignment, (a->>'mark')::int AS mark  
FROM enrollments e  
CROSS JOIN LATERAL jsonb\_array\_elements(e.grades->'assignments') a  
WHERE subject\_code = 'ML4001';

Explanation:

**e.grades->'assignments'**  
→ gets the value of the "assignments" key from the grades JSON.

**jsonb\_array\_elements(...) a**  
→ takes that JSON array and **unnests it into multiple rows**.

* Row 1: {"name":"A1","mark":40}
* Row 2: {"name":"A2","mark":45}

**CROSS JOIN LATERAL**  
→ means: *for each row in enrollments, run this function and join the results.*

* Without **LATERAL**, you can’t pass values from the left table (e.grades) into the function.

a->>'name'  
→ extracts "A1" / "A2" as text. (a->>'mark')::int  
→ extracts "40" / "45" as text, then casts to integer.

-- Extract project marks  
SELECT subject\_code, grades#>>'{project,title}' AS project\_title,  
 (grades#>>'{project,mark}')::int AS mark  
FROM enrollments  
WHERE grades ? 'project';

Explanation:

**WHERE grades ? 'project'**  
→ ensures we only look at rows where the grades JSON has a "project" key.

**grades#>>'{project,title}'**  
→ use **#>> (path operator)** to navigate nested JSON:

* Go into "project"
* Extract "title" as text

**(grades#>>'{project,mark}')::int**  
→ go into "project" → extract "mark" as text → cast to integer.

## 2.4 JSONPath Queries

A **JSON path** is like a *query language* (a bit like XPath for XML) that lets you navigate inside a JSON document.

* Think of a JSON document as a tree of objects and arrays.
* A JSON path is a string (starting with $) that says **“go here”** inside that tree.

Examples:

* $ → the root of the JSON document
* $.assignments → the assignments field
* $.assignments[\*].mark → all the mark values inside the assignments array
* $.\* → all the fields at the root, whatever their names

-- Students with any grade (for anything assignment, final etc) >= 85  
SELECT enrollment\_id, subject\_code  
FROM enrollments  
WHERE jsonb\_path\_exists(grades, '$.\* ? (@ >= 85)');

Explanation:

**jsonb\_path\_exists(grades, ...)**  
Checks if the JSON path finds at least one match inside the grades JSON. Returns true or false.

**Path: '$.\* ? (@ >= 85)'**

* $ = root of the JSON document (grades).
* .\* = all keys at the root (like "midterm", "final", "remarks", "assignments", "project", etc.).
* ? (@ >= 85) = filter: return only the values >= 85.

*-- For subject ML4001, show every assignment mark stored in the grades JSON.”*  
SELECT jsonb\_path\_query(grades, '$.assignments[\*].mark')  
FROM enrollments  
WHERE subject\_code = 'ML4001';

Explanation:

**jsonb\_path\_query(grades, ...)**  
Extracts the values that match the given path.

**Path: $.assignments[\*].mark**

* $ = root.
* .assignments = go into the assignments key.
* [\*] = all elements of the array.
* .mark = take the mark field of each.

## 2.5 Aggregations

-- Average final grade per university  
SELECT s.university, AVG((e.grades->>'final')::int)  
FROM students s  
JOIN enrollments e ON s.student\_id = e.student\_id  
WHERE e.grades ? 'final'  
GROUP BY s.university;

Explanation:

**JOIN**: links each enrollment (e) with the corresponding student (s) so we can access both the grades and the student’s university.

**e.grades ? 'final'**: only keep rows where the JSON grades has a key "final".

**e.grades->>'final'**: extract the final grade from JSON as text.

**::int**: cast it to an integer so math can be done.

**AVG(...)**: compute the average per group.

-- Best student per subject  
SELECT subject\_code, student\_id, MAX((grades->>'final')::int) AS best  
FROM enrollments  
WHERE grades ? 'final'  
GROUP BY subject\_code, student\_id;

Explanation:

**grades ? 'final'**: filter to rows where "final" exists.

**grades->>'final'**: extract the final grade (text).

**::int**: cast to integer.

**MAX(...)**: compute the maximum final grade.

**GROUP BY subject\_code, student\_id**: groups by subject and student.

## 2.6 Updates

-- Add exchange flag for all UCD students  
UPDATE students  
SET profile = profile || '{"exchange": false}'  
WHERE university = 'UCD';

Explanation:

**profile** is a JSONB column.

|| is the **concatenation / merge operator** for JSONB.

It merges the existing profile object with {"exchange": false}.

* If exchange already exists, it will be **overwritten** with false.
* If not, the key is added.

-- Update a nested grade

-- *For student 2 in DB4003, set their final grade inside the grades JSON to 90.*  
UPDATE enrollments  
SET grades = jsonb\_set(grades, '{final}', '90')  
WHERE subject\_code = 'DB4003' AND student\_id = 2;

Explanation:

* **jsonb\_set(target, path, new\_value)** replaces or inserts a value at the given path.
* grades is the JSONB column.
* '{final}' is the path (an array with one key, "final").
* '90' is the new value (a JSON number here, since no quotes inside).
* Only applies to the enrollment where subject = DB4003 and student\_id = 2.

-- Remove remarks for enrollments in subject DB4003  
UPDATE enrollments  
SET grades = grades - 'remarks'  
WHERE subject\_code = 'DB4003';

Explanation:

- 'key' removes a key from a JSONB object.

This removes the "remarks" field from the grades JSON.

Only for enrollments in DB4003.

**Exercise:**

1. Add a new key ECTS = 5 to all DB4003 enrollments.

2. Remove the key midterm where present.

## 2.8 Constraints

-- Ensure grades are JSON objects  
ALTER TABLE enrollments  
 ADD CONSTRAINT grades\_is\_object CHECK (jsonb\_typeof(grades) = 'object');  
  
-- Ensure final mark between 0–100  
ALTER TABLE enrollments  
 ADD CONSTRAINT final\_between CHECK ((grades ? 'final') IS NOT TRUE OR ((grades->>'final')::int BETWEEN 0 AND 100));

Explanation: Checking that either it doesn’t exist or that if it does that it is between 0 and 100

## 3. Exercises

1. Create a view subject\_results with student name, subject, year, final grade.

Create or Replace View….

1. Find top 3 students in CS4090 (by any grade).

For CS4090, students may have "project.mark" or "oral\_exam". We can take the \*\*maximum numeric value inside `grades`\*\* and rank by that.

Remember use Limit to limit your results

1. Using JSONPath, find students with assignment average > 40.

Join to students

CROSS JOIN LATERAL jsonb\_array\_elements(e.grades->'assignments') AS a

Remember to use Group and Having

1. Add a constraint ensuring that if oral\_exam exists, its value is ≤ 100.