FAKULTA ELEKTROTECHNICKÁ

České vysoké učení technické v Praze

B4M36DS2 – Database Systems 2

Practical Class 1

Introduction: Organization

Multi-Model Data & JSONB in PostgreSQL

Yuliia Prokop

prokoyul@fel.cvut.cz, Telegram **@Yulia_Prokop**

ČVUT FEL CourseWare Wiki

https://cw.fel.cvut.cz/b241/courses/b4m36ds2/start

Basic course information

Lectures: Monday, 9:15 – 10:45

Practical classes: Monday, 12:45 – 14:15, 14:30 – 16:00, 16:15 – 17:45

CourseWare Wiki – course materials

BRUTE – upload reports on the homework

NoSQL Server – submit and execute homework

Consultation – email me

Basic course information: assessment

	Maximum	Minimum required
Homework	42	25
Optional HW	18	0
Tests	10	5
Exam*	30	20
Total	100	50

* **Written exam** (mandatory) + **oral exam** (optional)

Lab accounts

Lab KN:E-307 user accounts

- Set your new password

  https://www.felk.cvut.cz/labpass/

NoSQL Server

NoSQL server

- SSH and SFTP access
- **nosql.felk.cvut.cz**
  - PostgreSQL, Redis, Cassandra, MongoDB, Neo4j, MapReduce
  - Java, Python
- Login and password: sent by e-mail

5

- To access the server outside the university, you need **FEL** or **CTU VPN.**

Vscode and JetBrains connections that do not work via sftp **are prohibited** due to extensive server disk space usage.
Here is an example of VSCode extension that uses SFTP and does not require a binary running on the server
https://marketplace.visualstudio.com/items?itemName=Kelvin.vscode-sshfs

Useful Tools

Linux

- **ssh** login@host – login to remote server
  ssh login**@nosql.felk.cvut.cz**
  exit
- **sftp** -P port login@host
  - **cd** directory – change remote directory
  - **lcd** directory – change local directory
  - **ls** – list remote directory contents
  - **lls** – list local directory contents
  - **put** local remote – copy a local file to the remote directory (sftp access)
  - **get** remote local –copy a remote to the local directory (sftp access)
  - **bye** or **exit** – disconnect

Useful Tools

Windows

- **PuTTY** – http://www.chiark.greenend.org.uk/~sgtatham/putty/
- **WinSCP** – http://winscp.net/

Firsts Steps

Change your initial password

- password
  - enter the current password
  - enter the new password

Browse important directories

- /home/login/ – personal directory with your data
- /home/DS2/ – shared directory with course data

Homework Assignments: submission

Submit your home assignments

- Upload your submission files to the **NoSQL server.**
- Put these files into a sub-directory **~/assignments/name/**, where **name** is the name of a given homework.
- This **name** parameter must also correspond to one of the predefined assignment names: **hw2**, **hw3**, **hw4**, **hw5**, **hw6**, **hw7**, **hw8** (case sensitive).
- Use **ssh** or **PuTTY** to open a remote shell connection to the NoSQL server.

Homework Assignments: submission

- Verify that everything is working as expected based on the instructions provided for a given homework assignment.
- Go to the ~/assignments/ directory and execute:

sudo submit_execute name

where name is the name of the homework

- Wait for the confirmation of success; otherwise, your homework will not be considered for submission.
- If any complications appear, write to me or send your solution by e-mail to prokoyul@fel.cvut.cz.
- Just for your convenience, you can check the submitted files in the ~/submissions/directory

Homework Assignments: submission

Example of submission

```
HOMEWORK ASSIGNMENT SUBMISSION SCRIPT
--------------------------------------------------------------------------------
Initializing assignment submission...
Preparing <sparql> assignment submission...
Submitting <sparql> assignment...
-rw-rw-r--  1 f221_serhii f221_serhii    0 Aug 28 18:40 data.ttl
-rw-rw-r--  1 f221_serhii f221_serhii    0 Aug 28 18:40 query1.sparql
-rw-rw-r--  1 f221_serhii f221_serhii    0 Aug 28 18:40 query2.sparql
-rw-rw-r--  1 f221_serhii f221_serhii    0 Aug 28 18:40 query3.sparql
-rw-rw-r--  1 f221_serhii f221_serhii    0 Aug 28 18:40 query4.sparql
-rw-rw-r--  1 f221_serhii f221_serhii    0 Aug 28 18:40 query5.sparql
Assignment <sparql> SUBMITTED by <f221_serhii> SUCCESSFULLY
Check your submission at <~/submissions/f221_serhii-sparql-20220828-184814>
--------------------------------------------------------------------------------
```

If one of the required files is absent:

```
f221_serhii@database:~/assignments$ sudo submit_execute cassandra
--------------------------------------------------------------------------------
HOMEWORK ASSIGNMENT SUBMISSION SCRIPT
--------------------------------------------------------------------------------
Initializing assignment submission...
Preparing <cassandra> assignment submission...
File <cassandra/script.cql> is required and missing
--------------------------------------------------------------------------------
```

Homework Assignments: submission

Requirements:

- Respect the prescribed names of individual files to be submitted (**case sensitive**)
- Place all the files in the root directory of your submission
- Do not include shared libraries or files that are not requested
- I.e. do not submit files that were not explicitly requested
- Do not redirect or suppress both standard and error outputs in your shell scripts
- All your files must be syntactically correct and executable without errors

Homework Assignments: upload to BRUTE

2. Upload files of your homework assignment to BRUTE

- Use the filename: **username_number.***, where the number is the homework number.
- The report must fulfill the requirements and have the following structure:
  - ✓ A verbal description of the task[13]
  - ✓ Code (copy from the file uploaded to the server)
  - ✓ Screenshot of code execution on the server
- If there are several problems, repeat the same for each of them

Dataset for examples – TMDB movie

We will use **TMDB movie and Oscar Nominees and Winners datasets** (from Kaggle)

Based on these datasets, **synthetic datasets** were generated.

.csv and .json files with data are available in the DS2 folder on the NoSQL server
Also, they are available on the course page

https://www.kaggle.com/datasets/tmdb/tmdb-movie-metadata/data

https://www.kaggle.com/datasets/viniciusno/oscar-nominees-and-winners-with-tmdb-metadata/data

Multi-Model Data & JSONB in PostgreSQL

Motivation

- Modern data often cannot be effectively modeled by only relational structures.
- PostgreSQL allows using both relational (tabular) and semi-structured (JSONB) data.
- Flexible models are powerful, but bring complexity and trade-offs.
- Goal: Learn to model, query, and critically compare relational and JSONB approaches in PostgreSQL for typical data science scenarios.

Multimodel Databases – Theoretical Background

Definition: Multimodel DBMSs support more than one data model

- e.g., relational + document

Advantages:

- Flexibility to mix schemas
- Ability to address diverse requirements within a single system

Challenges:

- Complexity of schema evolution
- Possible performance and integrity trade-offs

PostgreSQL:

- Primarily relational
- Powerful support for JSON/JSONB since v9.4
- Enables hybrid ("NoSQL in SQL") patterns

Multimodel Databases – Examples

	Relational DBMS	Key-value store	Document store	Wide column	Graph DBMS	RDF store	Spatial DBMS	Search engine	Time Series DBMS	Vector DBMS
Oracle	+		+		+	+	+			+
MySQL	+		+				+			
Microsoft SQL Server	+		+		+		+			
PostgreSQL	**+**		**+**		**+**		**+**			**+**
MongoDB			+				+	+	+	+
Redis		+	+		+		+	+	+	+
IBM Db2	+		+			+	+			
Elasticsearch			+				+	+		+
Apache Cassandra				+						+
Databricks	+		+							
MariaDB	+		+		+		+			
Amazon DynamoDB		+	+							

JSONB in PostgreSQL – Theory & Features

JSONB: Binary storage for JSON documents, allows indexing and efficient querying.

Use cases:

- Semi-structured or dynamic data
- Storing arrays, objects, nested fields
- Use when schema flexibility is needed

Key features:

- Supports GIN indexes for fast search
- JSONPath, operators (->, ->>, ?, etc.)
- Good for cases where attributes or structure vary between rows

Drawbacks:

- No foreign keys or referential integrity
- Complex queries are harder to optimize
- Performance drops with very large or deeply nested documents

JSONB documentation: https://www.postgresql.org/docs/current/datatype-json.html
https://www.postgresql.org/docs/9.5/functions-json.html

Assignment 1 – Model Transformation (Relational & JSONB)

Task

Transform a flat table of films and actors into two models:

- **Relational**

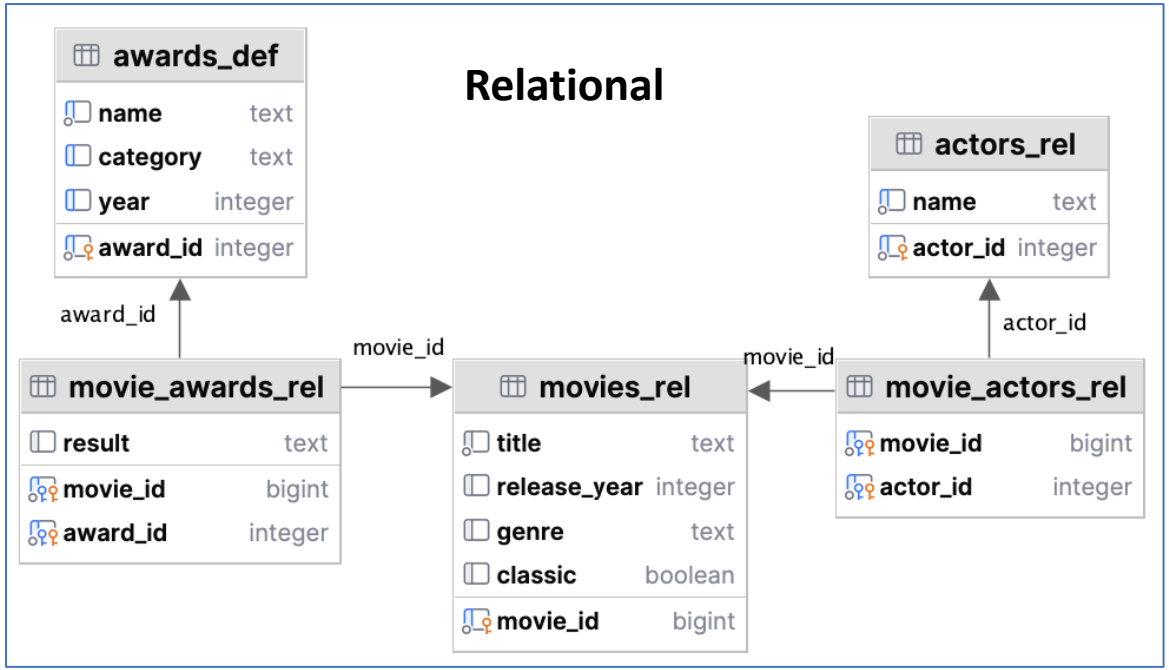
  Separate tables with many-to-many links between movies and awards

- **Hybrid (Relational** with **JSONB)**

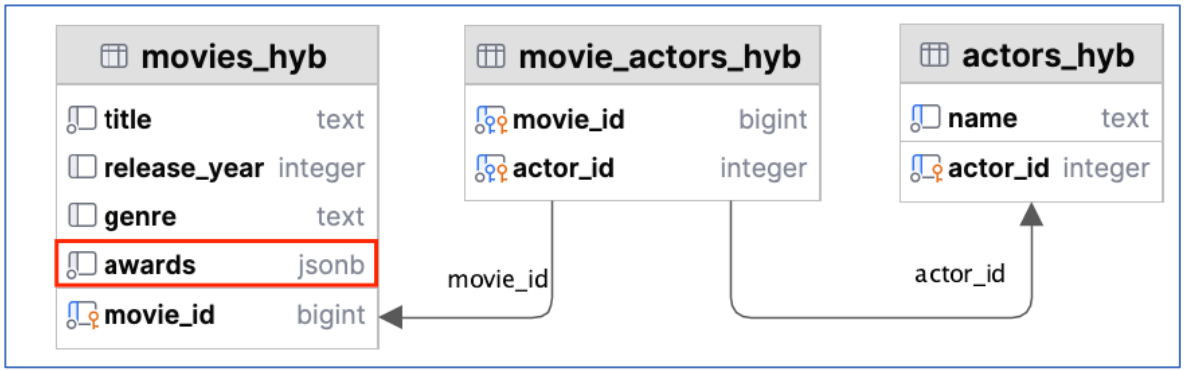
  Awards is not a table, but a document with nested fields in the movies table

Assignment 1 – Solution

Relational



Hybrid



Assignment 1 – Solution

Schema Definition: Relational

```
CREATE TABLE movies_rel (
  movie_id BIGINT PRIMARY KEY,
  title TEXT NOT NULL,
  release_year INT,
  genre TEXT
);

CREATE TABLE actors_rel (
  actor_id SERIAL PRIMARY KEY,
  name TEXT UNIQUE NOT NULL
);

CREATE TABLE movie_actors_rel (
  movie_id BIGINT REFERENCES movies_rel(movie_id),
  actor_id INT REFERENCES actors_rel(actor_id),
  PRIMARY KEY(movie_id,actor_id)
);
```

```
CREATE TABLE awards_def (
  award_id SERIAL PRIMARY KEY,
  name TEXT NOT NULL,
  category TEXT,
  year INT,
  UNIQUE(name,category,year)
);
CREATE TABLE movie_awards_rel (
  movie_id BIGINT REFERENCES movies_rel(movie_id),
  award_id INT REFERENCES awards_def(award_id),
  result TEXT,
  PRIMARY KEY(movie_id,award_id)
);
```

Assignment 1 – Solution

Schema Definition: Hybrid

```
CREATE TABLE movies_hyb (
  movie_id BIGINT PRIMARY KEY,
  title TEXT NOT NULL,
  release_year INT,
  genre TEXT,
  awards JSONB NOT NULL DEFAULT '[]'
);
CREATE TABLE actors_hyb (
  actor_id SERIAL PRIMARY KEY,
  name TEXT UNIQUE NOT NULL
);
CREATE TABLE movie_actors_hyb (
  movie_id BIGINT REFERENCES movies_hyb(movie_id),
  actor_id INT REFERENCES actors_hyb(actor_id),
  PRIMARY KEY(movie_id,actor_id)
);
```

Each movie has:

- standard relational attributes (title, year, genre, actors, etc.) and
- an **optional** awards field stored as JSONB.

This hybrid design showcases Postgres as a flexible **multi-model** system, blending SQL and NoSQL features in one schema.

Assignment 1 – Solution

Schema Definition: Hybrid (alternative version)

```
CREATE TABLE movies (
    movie_id SERIAL PRIMARY KEY,
    title TEXT,
    release_year INT,
    genre TEXT,
    language TEXT,
    actors TEXT[],          -- e.g. an array of actor names or separate M:N table
    awards JSONB            -- JSONB field for awards info (optional)
);
```

Data Characteristics — Semi-Structured Awards

- Awards entries are **optional** and **varied**: some records may be missing a year, category, or result.
- Enforcing a strict relational UNIQUE (name, category, **year)** may fail due to **missing or duplicate** fields.
- JSONB handles **missing keys** gracefully and preserves partial data without schema migration.

Examples of data insertion – Relational Model

```
-- Insert sample movie
INSERT INTO movies_rel VALUES (1, 'Inception', 2010, 'Sci-Fi');
-- Insert sample actor
INSERT INTO actors_rel (name) VALUES ('Leonardo DiCaprio');
-- Link actor to movie
INSERT INTO movie_actors_rel VALUES (
  1,
  (SELECT actor_id FROM actors_rel WHERE name='Leonardo DiCaprio')
);
-- Define awards
INSERT INTO awards_def (name, category, year) VALUES ('Oscar', 'Best Picture', 2011);
-- Associate award with movie
INSERT INTO movie_awards_rel (movie_id, award_id, result)
VALUES (
  1,
  (SELECT award_id FROM awards_def WHERE name='Oscar' AND category='Best Picture'
AND year=2011),
  'won'
);
```

Examples of data insertion – Hybrid Model

```
-- Insert sample movie with embedded awards
INSERT INTO movies_hyb (movie_id, title, release_year, genre, awards)
VALUES (
  1, 'Inception', 2010, 'Sci-Fi',
  '[
    {"award":"Oscar","category":"Best Picture","year":2011,"result":"won"}
  ]'
);

-- Insert actor
INSERT INTO actors_hyb (name) VALUES ('Leonardo DiCaprio');

-- Link actor to movie
INSERT INTO movie_actors_hyb VALUES (
  1,
  (SELECT actor_id FROM actors_hyb WHERE name='Leonardo DiCaprio')
);
```

Assignment 2 – Querying Nested JSONB vs Relational Data

Task

List movie titles that have won an "Oscar” award in 2002.

- For relational:

  join movies_rel → movie_awards_rel → awards_def

- For hybrid:

  search awards JSONB array in movies_hyb.

Assignment 2 – Solution

Relational

```
SELECT title
  FROM movies_rel m
  JOIN movie_awards_rel mar USING(movie_id)
  JOIN awards_def ad USING(award_id)
WHERE name = 'Oscar' AND result = 'winner';
```

Hybrid

```
SELECT title
  FROM movies_hyb
 WHERE EXISTS (
   SELECT 1 FROM jsonb_array_elements(awards) a
     WHERE a->>'award' = 'Oscar' AND a->>'result' = 'winner'
 );
```

Assignment 2a – Querying Nested JSONB vs Relational Data

Task

Find all movies that won an Oscar for Best Actress in 1993.

Assignment 2 – Solution

Relational

```
SELECT title, release_year, name, year, category, result
FROM movies_rel
   JOIN movie_awards_rel USING (movie_id)
       JOIN awards_def USING (award_id)
WHERE name = 'Oscar'
  AND category = 'Best Actress'
  AND year = 1993
  AND result = 'winner'
ORDER BY year;
```

Hybrid

```
SELECT title, release_year
FROM movies_hyb
WHERE awards @> '[{"name": "Oscar",
                  "category": "Best Actress",
                  "year": 1993,
                  "result": "winner"}]';
```

```
SELECT title
FROM movies_hyb
WHERE EXISTS (
   SELECT 1
   FROM jsonb_array_elements(movies_hyb.awards) AS award
   WHERE award->>'name' = 'Oscar'
     AND award->>'category' = 'Best Actress'
     AND award->>'year' = '1993'
     AND award->>'result' = 'winner'
);
```

Assignment 3 – Count Awards per Movie

Task

Count the number of awards each movie has received.

- For relational:

  aggregate rows in movie_awards_rel

- For hybrid:

  use jsonb_array_length(awards).

Assignment 3 – Solution

Relational

```
SELECT title, COUNT(*) AS award_count
FROM movies_rel
   JOIN movie_awards_rel mar USING(movie_id)
       JOIN awards_def USING (award_id)
WHERE result = 'winner'
GROUP BY title
ORDER BY award_count DESC
LIMIT 15;
```

Hybrid

```
SELECT title, COUNT(*) AS awards_count
FROM movies_hyb,
    jsonb_array_elements(awards) AS award
WHERE award->>'result' = 'winner'
GROUP BY title
ORDER BY awards_count DESC
LIMIT 15;
```

Assignment 4 – Find Movies Missing Award Year

Task

Count the number of awards each movie has received.

- For relational:

  aggregate rows in movie_awards_rel

- For hybrid:

  use jsonb_array_length(awards).

Assignment 4 – Solution

Relational

```
SELECT DISTINCT m.title
  FROM movies_rel m
  JOIN movie_awards_rel mar ON m.movie_id = mar.movie_id
  JOIN awards_def ad ON mar.award_id = ad.award_id
 WHERE ad.year IS NULL;
```

Hybrid

```
SELECT title
  FROM movies_hyb
 WHERE EXISTS (
   SELECT 1 FROM jsonb_array_elements(awards) a
    WHERE NOT (a ? 'year')
 );
```

Assignment 5 – Add “Classic” Flag

Task

Mark movies as “classic” if release_year < 1980.

- For relational:

  alter table + bulk update

- For hybrid:

  embed flag via JSONB functions.

Assignment 5 – Solution

Relational

```
ALTER TABLE movies_rel ADD COLUMN classic BOOLEAN;
UPDATE movies_rel SET classic = (release_year < 1980);
```

Hybrid

```
-- This fails because awards is a JSON array
UPDATE movies_hyb
  SET awards = jsonb_set(awards, '{classic}', to_jsonb(release_year < 1980));
```

Error: path element at position 1 is not an integer: "classic"

```
-- Wrap array into an object to add a field
UPDATE movies_hyb
  SET awards = (
    jsonb_set(
      jsonb_build_object('awards', awards),
      '{classic}',
      to_jsonb(release_year < 1980)
    ) -> 'awards'
  );
```

Assignment 6 – Indexing & Performance

How can we optimize queries on awards data in both models?

What do the execution plans look like?

Add indexes to speed up queries:

- an index on the awards table's columns VS a GIN index on the JSONB column
- use **EXPLAIN ANALYZE** to compare query performance.

Assignment 6 – Solution

Hybrid

```
CREATE INDEX idx_awards_jsonb ON movies_hyb USING GIN (awards);

EXPLAIN ANALYZE
SELECT title
FROM movies_hyb
WHERE awards @> '[{"award": "Oscar", "category": "Best Actress", "year": 1993}]';
```

QUERY PLAN
Bitmap Heap Scan on movies_hyb (cost=30.60..34.61 rows=1 width=16) (actual time=0.221..0.222 rows=0 loops=1)
Recheck Cond: (awards @> '[{"year": 1993, "award": "Oscar", "category": "Best Actress"}]'::jsonb)
-> Bitmap Index Scan on idx_awards_jsonb (cost=0.00..30.60 rows=1 width=0) (actual time=0.211..0.212 rows=0 loops=1)
Index Cond: (awards @> '[{"year": 1993, "award": "Oscar", "category": "Best Actress"}]'::jsonb)
Planning Time: 0.862 ms
Execution Time: 0.299 ms

Assignment 6 – Solution

Relational

```
CREATE INDEX idx_awards
   ON awards_def(name, category, year);

CREATE INDEX idx_movie_awards_rel_award_result
   ON movie_awards_rel (award_id, result);

EXPLAIN ANALYZE
SELECT title
FROM movies_rel
       JOIN movie_awards_rel mar USING(movie_id)
       JOIN awards_def USING (award_id)
WHERE name='Oscar' AND category='Best Actress'
 AND year=1993 AND result='winner';
```

Assignment 6 – Solution

Relational

```
QUERY PLAN
Nested Loop  (cost=4.88..26.56 rows=4 width=16) (actual time=0.209..0.252 rows=3 loops=1)
  ->  Nested Loop  (cost=4.60..24.72 rows=4 width=8) (actual time=0.196..0.206 rows=3 loops=1)
        ->  Index Scan using awards_def_name_category_year_key on awards_def  (cost=0.27..8.29 rows=1 width=4) (actual time=0.107..0.109 rows=1 loops=1)
              Index Cond: ((name = 'Oscar'::text) AND (category = 'Best Actress'::text) AND (year = 1993))
        ->  Bitmap Heap Scan on movie_awards_rel mar  (cost=4.32..16.38 rows=4 width=12) (actual time=0.079..0.086 rows=3 loops=1)
              Recheck Cond: ((award_id = awards_def.award_id) AND (result = 'winner'::text))
              Heap Blocks: exact=3
              ->  Bitmap Index Scan on idx_movie_awards_rel_award_result  (cost=0.00..4.32 rows=4 width=0) (actual time=0.059..0.059 rows=3 loops=1)
                    Index Cond: ((award_id = awards_def.award_id) AND (result = 'winner'::text))
  ->  Index Scan using movies_rel_pkey on movies_rel  (cost=0.28..0.46 rows=1 width=24) (actual time=0.013..0.013 rows=1 loops=3)
        Index Cond: (movie_id = mar.movie_id)
Planning Time: 1.184 ms
Execution Time: 0.313 ms
```

Hybrid

```
QUERY PLAN
Bitmap Heap Scan on movies_hyb  (cost=39.43..43.44 rows=1 width=16) (actual time=0.158..0.159 rows=0 loops=1)
  Recheck Cond: (awards @> '[{"year": 1993, "award": "Oscar", "result": "winner", "category": "Best Actress"}]'::jsonb)
  ->  Bitmap Index Scan on idx_awards_jsonb  (cost=0.00..39.43 rows=1 width=0) (actual time=0.155..0.155 rows=0 loops=1)
        Index Cond: (awards @> '[{"year": 1993, "award": "Oscar", "result": "winner", "category": "Best Actress"}]'::jsonb)
Planning Time: 0.716 ms
Execution Time: 0.240 ms
```

Assignment 6a – Indexing & Performance

Task

Find distinct movie titles that won awards between 2019 and 2020.

Assignment 6a – Solution

Relational

```
CREATE INDEX idx_awards_year
  ON awards_def USING btree (year);

CREATE INDEX idx_mar_award
  ON movie_awards_rel USING btree (award_id);

CREATE INDEX idx_mar_movie
  ON movie_awards_rel USING btree (movie_id);

EXPLAIN (ANALYZE, BUFFERS)
SELECT DISTINCT title
FROM awards_def
        JOIN movie_awards_rel USING (award_id)
                JOIN movies_rel USING (movie_id)
WHERE year BETWEEN 2019 AND 2020;
```

Relational Model benefits from **B-tree** indexes on scalar columns, resulting in index scans and minimal I/O.

Assignment 6a – Solution

```
1  HashAggregate  (cost=711.16..718.20 rows=704 width=21) (actual time=8.571..8.772 rows=774 loops=1)
2    Group Key: m.title
3    Batches: 1  Memory Usage: 105kB
4    Buffers: shared hit=3041
5    ->  Nested Loop  (cost=7.94..709.40 rows=704 width=21) (actual time=0.140..7.820 rows=789 loops=1)
6          Buffers: shared hit=3041
7          ->  Nested Loop  (cost=7.65..461.37 rows=704 width=8) (actual time=0.093..2.928 rows=789 loops=1)
8                Buffers: shared hit=674
9                ->  Bitmap Heap Scan on awards_def a  (cost=4.34..7.63 rows=19 width=4) (actual time=0.047..0.085 rows=18 loops=1)
10                     Recheck Cond: ((year >= 2019) AND (year <= 2020))
11                     Heap Blocks: exact=3
12                     Buffers: shared hit=4
13                     ->  Bitmap Index Scan on idx_awards_year  (cost=0.00..4.34 rows=19 width=0) (actual time=0.035..0.036 rows=18 loops=1)
14                           Index Cond: ((year >= 2019) AND (year <= 2020))
15                           Buffers: shared hit=1
16               ->  Bitmap Heap Scan on movie_awards_rel mar  (cost=3.31..23.51 rows=37 width=12) (actual time=0.029..0.138 rows=44 loops=18)
17                     Recheck Cond: (a.award_id = award_id)
18                     Heap Blocks: exact=634
19                     Buffers: shared hit=670
20                     ->  Bitmap Index Scan on idx_mar_award  (cost=0.00..3.30 rows=37 width=0) (actual time=0.015..0.015 rows=44 loops=18)
21                           Index Cond: (award_id = a.award_id)
22                           Buffers: shared hit=36
23         ->  Index Scan using movies_rel_pkey on movies_rel m  (cost=0.29..0.35 rows=1 width=29) (actual time=0.005..0.005 rows=1 loops=789)
24               Index Cond: (movie_id = mar.movie_id)
25               Buffers: shared hit=2367
26 Planning:
27   Buffers: shared hit=28
28 Planning Time: 1.756 ms
29 Execution Time: 9.064 ms
```

29 rows

Assignment 6a – Solution

Hybrid

GIN index (general JSONB containment)

```
CREATE INDEX idx_movies_hyb_awards_gin
  ON movies_hyb USING gin (awards jsonb_path_ops);

EXPLAIN (ANALYZE, BUFFERS)
SELECT title
FROM movies_hyb
WHERE EXISTS (
  SELECT 1
  FROM jsonb_array_elements(awards) AS elem
  WHERE (elem->>'year')::int BETWEEN 2019 AND 2020
);
```

Hybrid Model with generic GIN index speeds up containment (@>) checks, but not arbitrary JSONB operations (like jsonb_array_elements or length filters).

Assignment 6a – Solution

	QUERY PLAN
1	Seq Scan on movies_hyb (cost=10000000000.00..10000000645.11 rows=4803 width=21) (actual time=77.174..85.222 rows=774 loops=1)
2	Filter: jsonb_path_exists(awards, '$[*]?(@."year" >= 2019 && @."year" <= 2020)'::jsonpath, '{}'::jsonb, false)
3	Rows Removed by Filter: 13635
4	Buffers: shared hit=465
5	Planning:
6	Buffers: shared hit=19
7	Planning Time: 0.334 ms
8	JIT:
9	Functions: 4
10	Options: Inlining true, Optimization true, Expressions true, Deforming true
11	Timing: Generation 3.445 ms, Inlining 17.793 ms, Optimization 37.496 ms, Emission 20.373 ms, Total 79.106 ms
12	Execution Time: 88.885 ms

Assignment 6a – Solution

Hybrid

JSONPath predicate index (range on year)

```
CREATE INDEX idx_hyb_awards_year_path
  ON movies_hyb USING gin (awards jsonb_path_ops)
  WHERE jsonb_path_exists(
    awards,
    '$[*] ? (@.year >= 2019 && @.year <= 2020)'::jsonpath
  );

EXPLAIN (ANALYZE, BUFFERS)
SELECT title
FROM movies_hyb
WHERE EXISTS (
  SELECT 1
  FROM jsonb_array_elements(awards) AS elem
  WHERE (elem->>'year')::int BETWEEN 2019 AND 2020
);
```

A **JSONPath index** can target specific patterns inside the JSONB, narrowing Seq Scans to Bitmap Index Scans for path queries.

Assignment 6a – Solution

```
QUERY PLAN
Bitmap Heap Scan on movies_hyb  (cost=273.20..798.24 rows=4803 width=21) (actual time=0.359..0.813 rows=774 loops=1)
  Recheck Cond: jsonb_path_exists(awards, '$[*]?(@."year" >= 2019 && @."year" <= 2020)'::jsonpath, '{}'::jsonb, false)
  Heap Blocks: exact=282
  Buffers: shared hit=287
  ->  Bitmap Index Scan on idx_hyb_awards_year_path  (cost=0.00..272.00 rows=4803 width=0) (actual time=0.317..0.317 rows=774 loops=1)
        Buffers: shared hit=5
Planning:
  Buffers: shared hit=19
Planning Time: 0.240 ms
Execution Time: 0.883 ms
```

1.Specialized JSONPath Index perfectly matches the filter year BETWEEN 2019 AND 2020, so the planner only reads the rows that match exactly.
2.Maintenance Overhead: Each INSERT/UPDATE/DELETE in movies_hyb must also update this conditional index, which can slow write operations on large JSONB data.
3.Flexibility vs. Performance

1. A **generic GIN index** is versatile but slow for range queries inside JSONB.
2. A **JSONPath predicate index** is extremely fast but only for the specific JSON path. Other JSON queries (e.g., filtering by category) would require additional indexes.

4.Version Dependency JSONPath indexes are supported only in PostgreSQL 12+; older versions cannot leverage this optimization.

Reflection & Best Practices

Relational

strong integrity, optimized joins, clear schema

Hybrid JSONB

flexible, easy to evolve, embed optional fields

Strategy:

keep core entities relational; embed only truly variable metadata

Comparison – Modeling Pros & Cons

Aspect	Relational (Separate awards table)	JSONB Hybrid (awards JSONB column)
Schema	Explicit schema for awards (fixed columns, data types)	Flexible schema, can vary per movie; no upfront design for fields
Optional Data	Movies without awards simply have no related rows (natural)	Movies without awards can have NULL/empty JSON (natural fit)
Complex Structure	If awards have many attributes or types, they require multiple columns or tables. Can become complex	Can store complex, nested award data in one JSON field. Simpler to represent hierarchical data
Ease of Change	Changing the award structure needs ALTER TABLE (adding columns, etc.) and possibly backfilling data	Changing structure is easy – just start adding new JSON keys/values; no table migration needed
Data Integrity	Enforces each award entry has the defined columns (though some could be NULL). Foreign key ensures awards link to a valid movie	Lacks inherent constraints inside JSON (no FK or per-key type enforcement by default). Requires application or CHECK constraints to ensure the presence of keys if needed
Normalization	No duplication of award info; can reference an award type dictionary if needed. Structured for queries (e.g., easily count all “Oscar” awards)	Denormalized in each movie document; keys like "award" are repeated in each entry. Slightly larger storage overhead due to repeated field names