NDBI040: PRACTICAL CLASS 7

POSTGRESQL

Tutor: Pavel Čontoš; December 2nd 2020

(RECOMMENDED) REQUIREMENTS

- Database concepts
- Advanced knowledge of SQL

- NetBeans IDE or pgAdmin or psql
- macOS / Linux command line or PuTTy / WinSCP on Windows

SERVER ACCESS

CONNECT TO NOSQL SERVER

- `ssh` on macOS / Linux
- `PuTTy` on Windows

- nosql.ms.mff.cuni.cz:42222
- Login and password send by e-mail
- Change your initial password (if not yet changed) by `passwd`

TRANSFER FILES

- `scp` on macOS / Linux
- `WinSCP` on Windows

POSTGRESQL

- Widely used open source multi-model DBMS*
  - Originally relational database (relational model)
  - Secondary document store (XML, JSON, TextSearch)

- Supports native XML
  - SQL/XML
  - https://www.postgresql.org/docs/11/datatype-xml.html
- Supports native JSON since 9.2 (2012)
  - SQL/JSON
  - SQL/JSON Path Language added in version 12 (out of scope of practical class)
- Supports Text Search
  - https://www.postgresql.org/docs/11/textsearch.html

* https://db-engines.com/en/system/PostgreSQL

POSTGRESQL

START POSTGRESQL SHELL

- `psql`

BASIC COMMANDS

- `\?`
  - Displays a brief description
- `\q`
  - Closes the current connection
- `\l[+]`
  - Lists all databases in the current server (with additional information)
- `\c m201_student`
  - Connect to the appropriate database

DATA TYPES: JSON, JSONB

- JSON-specific functions and operators for JSON, JSONB data types

JSON

- `Exact copy` of the input text as JSON
- Processing functions must reparse JSON on every execution
- Preserves the order of object keys, keeps duplicate object keys

JSONB

- Stored in decomposed `binary format` (slower saving due to conversion overhead)
- Faster to process, no reparsing is needed
- Additional functions and operators are provided (i.e. `comparison operators` <, >, <=, >=, =, <>, !=)
- Supports `indexing`

EXERCISE 1: DATA TYPES (SOLVED)

- Compare JSON and JSONB data types

- ```
  SELECT '{           "title": { "cs": "Samotari", "en": "Loners" }, "year": 2.0e+3, "rating": 84, "length": 103, "actors": [ "trojan", "machacek", "schneiderova" ], "genres": [ "comedy", "drama" ], "country": [ "CZ", "SI" ] }'::json;
  ```

- ```
  SELECT '{ "title": { "cs": "Samotari", "en": "Loners" }, "year": 2.0e+3, "rating": 84, "length": 103, "actors": [ "trojan", "machacek", "schneiderova" ], "genres": [ "comedy", "drama" ], "country": [ "CZ", "SI" ] }'::jsonb;
  ```

EXERCISE 2: CREATE TABLE

- Create a new table for actors
  - Columns: text identifier (`id`) , json data (`data`), text array movies (`movies`)
- Create a new table for movies
  - Columns: text identifier (`id`), jsonb data (`data`)

BROWSE EXISTING TABLES

- `\dt[+]`
  - Lists all tables in database (with additional information)
- `\d[+] TABLE_NAME`
  - Describes a table (with additional information)

INSERT

- Insert data about movies into table `movies`

- ```
  INSERT INTO movies (id,data) VALUES ('medvidek','{ "title" : "Medvidek", "year": 2007,
  "rating": 53, "length": 100 }');
  ```

- ```
  INSERT INTO movies (id,data) VALUES ('zelary','{ "title": "Zelary", "year": 2003,
  "rating":81, "length":142, "actors": [ ], "genres": [ "romance", "drama" ] }');
  ```

- ```
  INSERT INTO movies (id,data) VALUES ('kolja','{ "title": "Kolja", "year": 1996,
  "rating":86, "length":105, "awards": [ { "type": "Czech Lion", "year": 1996 },
  { "type": "Academy Awards", "category": "A", "year": 1996 } ] }');
  ```
  ```

UPDATE AND DELETE (JSONB)

- Operator ||
  - Concatenates two jsonb values into a new jsonb value
  - `UPDATE movies SET data = data || '{"id": "medvidek" }' WHERE id = 'medvidek';`

- Operator -
  - Deletes (multiple) key/value pair(s), string elements or array element
  - `UPDATE movies SET data = data - 'id' WHERE id = 'medvidek';`
  - `UPDATE movies SET data = data - ARRAY['actors','genres'] WHERE id = 'zelary';`

- Operator #-
  - Deletes the field or element with specified path
  - `UPDATE movies SET data = data #- '{awards,0}' WHERE id = 'kolja';`

INSERT SAMPLE DATA

- First, delete existing data
  - `DELETE FROM movies;`

- Download file `data.txt` from the practical class website and insert sample data to your database

JSON AND JSONB OPERATORS

- Operator `->`
  - Get JSON object field by key or JSON array element (indexed from 0, negative integers count from the end)
  - Allows operator chaining, e.g. `DOCUMENT -> 'A' -> 1 -> 'C'`
  - `SELECT data -> 'name' AS name FROM actors;`

- Operator `->>`
  - Get JSON array element or JSON object field as text
  - `SELECT data ->> 'name' AS name FROM actors;`

- Chaining:
  - `SELECT * FROM actors WHERE data -> 'name' ->> 'first' = 'Ivan';`
  - `SELECT data ->> 'name' ->> 'last' AS lastname FROM actors;`

JSON AND JSONB OPERATORS

- Operator #>
  - Get JSON object at specified path
  - `SELECT data #> '{awards,0}' -> 'type' AS award FROM movies;`

- Operator #>>
  - Get JSON object at specified path as text
  - `SELECT data #>> '{actors,1}' AS name FROM movies;`

ADDITIONAL JSONB OPERATORS

- Operator @>
  - Test whether left JSON value contains the right JSON path/value at the top level
  - `SELECT data -> 'title' FROM movies WHERE data @> '{"length" : 100}';`

- Operator <@
  - Test whether are left JSON path/value entries contained at the top level within the right JSON value
  - `SELECT data -> 'title' FROM movies WHERE '{"length" : 103}' <@ data;`

- Operator ?
  - Test whether the string exists as a top-level key within the JSON value
  - `SELECT data -> 'title' FROM movies WHERE data ? 'awards';`

ADDITIONAL JSONB OPERATORS

- Operator ?|
  - Test whether any of the string values exist as a top-level keys
  - `SELECT data->'title' AS title FROM movies WHERE data ?| ARRAY['awards','actors'];`

- Operator ?&
  - Test whether all of the string values exist as a top-level keys
  - `SELECT data->'title' AS title FROM movies WHERE data ?& ARRAY['awards','actors'];`

TYPE CASTS

- Specifies a conversion from one data type to another
  - `CAST ( expression AS type )`
  - `expression::type`

```
SELECT data -> 'length' FROM movies WHERE data ->> 'length' > '100';
SELECT data -> 'length' FROM movies WHERE (data ->> 'length')::INTEGER > 100;
```

JSON CREATION FUNCTIONS

- `to_json[b](anyelement)`
  - Returns the value as JSON or JSONB
  - `SELECT to_jsonb(movies) AS movies_json FROM actors;`

- `array_to_json(anyarray [, pretty_bool])`
  - Returns the array as a JSON array
  - `SELECT array_to_json(movies, true) AS movies_json FROM actors;`

- `row_to_json(record [, pretty_bool])`
  - Returns the row as a JSON object
  - `SELECT row_to_json(row, true) FROM (SELECT * FROM actors) row;`

JSON CREATION FUNCTIONS

- `json[b]_object(text[])`
  - Builds a JSON object out of a text array.
  - `SELECT jsonb_object(ARRAY['id',id,'type', data->>'actors']) FROM movies;`

- `json[b]_object(keys text[], values text[])`
  - Builds a JSON object out of keys and values from two separate arrays.
  - `SELECT jsonb_object(ARRAY['id','actors'], ARRAY[id, data->>'actors']) FROM movies;`

- ...

JSON PROCESSING FUNCTIONS

- `json[b]_array_length(json[b])`
  - Returns the number of elements in the outermost JSON array
  - `SELECT data FROM movies WHERE jsonb_array_length(data -> 'actors') > 3;`

- `json[b]_each(json[b])`
- `json[b]_each_text(json[b])`
  - Expands the outermost JSON object into a set of key/value pairs. The returned values can be represented as a text
  - `SELECT json_each(data) FROM actors;`
  - `SELECT jsonb_each_text(data) FROM movies;`

- `json[b]_object_keys(json[b])`
  - Returns set of keys in the outermost JSON object
  - `SELECT jsonb_object_keys(data) FROM movies WHERE id = 'kolja';`

JSON PROCESSING FUNCTIONS

- `json[b]_array_elements(json[b])`
- `json[b]_array_elements_text(json[b])`
  - Expands a JSON array to a set of JSON (or text) values
  - `SELECT jsonb_array_elements(data -> 'actors') FROM movies WHERE id = 'medvidek';`

- `json[b]_typeof(json[b])`
  - Returns the type (object, array, string, number, boolean, and null) of the outermost JSON value as a text string
  - `SELECT id, jsonb_typeof(data -> 'title') FROM movies;`

JSON PROCESSING FUNCTIONS

- `jsonb_set(target jsonb, path text[], new_value jsonb [, create_missing boolean])`
  - Replaces a value inside a JSON at the defined position
  - `UPDATE movies SET data=jsonb_set(data, '{actors,1}', '"geislerova"') WHERE data->>'title'='Medvidek';`
- `jsonb_insert(target jsonb, path text[], new_value jsonb [, insert_after boolean])`
  - Inserts a value to a JSON at the defined position
  - `UPDATE movies SET data=jsonb_insert(data, '{actors,1}', '"machacek"') WHERE data->>'title'='Medvidek';`
- `jsonb_pretty(from_json jsonb)`
  - Returns JSON as intended JSON text
  - `SELECT jsonb_pretty(data) FROM movies;`
- ...

AGGREGATE FUNCTIONS

- `min(expression)`
  - Minimum value of expression across all non-null input values
- `max(expression)`
  - Maximum value of expression across all non-null input values
- `avg(expression)`
  - The arithmetic mean of all non-null input values
- `sum(expression)`
  - Sum of expression across all non-null input values
- `count(expression)`
  - Number of input rows for which the value of expression is not null, e.g. count(*)
- `every(expression)`
  - True if all input values are true, otherwise false
- ...

EXERCISE 3: AGGREGATE FUNCTIONS (SOLVED)

- Determine minimal, maximal, average and sum of lengths of movies, count movies in database and decide whether all the movies are longer than 100 minutes.

```
SELECT
  MIN ((data ->> 'length')::INTEGER) AS min_length,
  MAX ((data ->> 'length')::INTEGER) AS max_length,
  AVG ((data ->> 'length')::INTEGER) AS average_length,
  SUM ((data ->> 'length')::INTEGER) AS sum_length,
  COUNT (data ->> 'length') AS count_movies,
  EVERY ((data ->> 'length')::INTEGER > 100) AS all_long
FROM movies;
```

JSONB INDEXING

- Improves query speed
  - `CREATE INDEX movies_index ON movies ((data->>'year'));`
- Multi-column indexes are allowed
  - `CREATE INDEX actors_index ON actors ((data -> 'name' ->> 'last'), (data ->> 'year') DESC);`

- `\d movies`
- `\d actors`

EXERCISE 4

- Find all actors born after year 1966
  - Sort actors from youngest to oldest
  - Return only actor name and surname

EXERCISE 5

- Find all movies having Czech and English titles
  - Return Czech and English title only

EXERCISE 6

- Find all movies that are comedies and dramas at the same time or have a rating 80 or more
  - Sort result according to theirs rating in descending order
  - Return movie title, genres and rating

EXERCISE 7

- Find all movies that were awarded by Czech Lion
  - Return title of the movie

EXERCISE 8

- Find movies filmed between years 2000 and 2006 such that they have a director specified
  - Sort result according to year in descending order
  - Return only movie title, year and director

EXERCISE 9

- Find movie with Czech title equal to "Vratne lahve".
  - Note that property title may be either string or object having nested properties
  - Return only title

EXERCISE 10

- Find all movies that have more than average number of actors across all the movies
  - Return movie title and number of actors

EXERCISE 11

- Find all movies filmed after 2000
  - Sort movies according to number of actors in descending order
  - Return movie title, year and number of actors as a JSON
  - Apply pretty print

REFERENCES

- PostgreSQL
  - https://www.postgresql.org
- JSON Types
  - https://www.postgresql.org/docs/current/datatype-json.html
- JSON Functions and Operators
  - https://www.postgresql.org/docs/11/functions-json.html
- Aggregate Functions
  - https://www.postgresql.org/docs/11/functions-aggregate.html