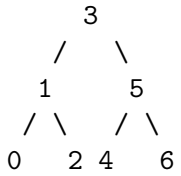
Databases
TDA357/DIT621

Exercise 5

JSON

1 Binary Trees

A binary tree is a tree where every node either branches into two binary trees or is a leaf, i.e. contains just a value. Here is an example of a binary tree:



1. Design a JSON schema for representing binary trees using key/values for children.

   Both the branching nodes and leaves should carry integer values. An important property is that each node in the tree either has both a left and a right subtree, or no subtree at all (that is, not just a left subtree).

   For this schema:

   (a) Show a JSON element representing the tree above, and that is valid according to your schema.

   (b) Write a (Postgres) JSON Path that returns:

   i. All values in the binary tree; for the above example, it should return 0,1,2,3,4,5,6 (in any order).

   ii. All data values of left subtrees (0,1,4).

   iii. All values in the third level of the tree (0,2,4,6).

   iv. All values greater than 3.

   v. All values greater than the value in the root node (should work for all trees).

   (c) Try to even define a variant where data is required in leaves but not in branches.

2. Do the same as above (schema, element and paths) but for a representation using arrays.

Solution:

1. Schema using key/values:

```
{"type": "object",
 "oneOf": [{"$ref": "#/definitions/leaf"},
           {"$ref": "#/definitions/tree"}],
 "definitions": {
    "leaf": {
      "type": "object",
      "properties": {"value": {"type": "integer"}},
      "required": ["value"],
      "additonalProperties": false},
    "tree": {
      "type": "object",
      "properties": {
        "value": {"type": "integer"},
        "left": {"$ref": "#"},
        "right": {"$ref": "#"}},
      "required": ["value","left","right"],
      "additonalProperties": false}
 }
}
```

   (a) Element:

```
{ "value":3,
  "left": {
    "value":1,
    "left": {"value":0},
    "right": {"value":2}
  },
  "right": {
    "value":5,
    "left": {"value":4},
    "right": {"value":6}
  }
}
```

(b) Paths:

```
WITH Data AS
  (SELECT '{ "value":3,
             "left": { "value":1,
                       "left": {"value":0},
                       "right": {"value":2}
                     },
             "right": { "value":5,
                        "left": {"value":4},
                        "right": {"value":6}
                      }
           }' :: JSONB AS val)
SELECT jsonb_path_query (val,
                         'strict $')  -- here you paste your query
FROM Data AS res;         -- like this it will return the whole object
```

i. `'strict $.**.value'`

ii. `'strict $.**.left.value'`

iii. `'lax $.*.*.value'`

iv. `'strict $.**?(@ > 3)'`
   alternative: `'strict $.**.value?(@ > 3)'`
   alternative: `'strict $.**?(@.value > 3).value'`

v. `'strict $.**?(@ > $.value)'`
   alternative: `'strict $.**.value?(@ > $.value)'`
   alternative: `'strict $.**?(@.value > $.value).value'`

(c) Variant schema: just remove required "value" from what is required in "tree".

2. Schema using arrays:

```
{ "type": "object",
  "properties": {
    "value": {"type": "integer"},
    "children": {"type": "array",
                 "items" : {"$ref": "#"},
                 "minItems":2,
                 "maxItems":2}},
  "required": ["value"],
  "additionalProperties": false
}
```

(a) Element:

```
{ "value":3,
  "children": [
    {"value":1,
     "children": [{"value":0},{"value":2}]},
    {"value":5,
     "children": [{"value":4},{"value":6}]}
  ]
}
```

(b) Paths:

i. `'strict $.**.value'`

ii. `'strict $.**.children[0].value'`

iii. `'strict $.children[*].children[*].value'`

iv. `'strict $.**?(@ > 3)'`
alternative: `'strict $.**.value?(@ > 3)'`
alternative: `'strict $.**?(@.value > 3).value'`

v. `'strict $.**?(@ > $.value)'`
alternative: `'strict $.**.value?(@ > $.value)'`
alternative: `'strict $.**?(@.value > $.value).value'`

(c) Variant schema: kind of tricky. Requires a separate definitions for branches/leafs, with leafs requiring "value" and branches requiring "children" (and having data optional).

2 Flights

Given the following schema:

Airports (<u>code</u>, city)

FlightCodes (<u>code</u>, airlineName)

Flights (departureAirport, destinationAirport, <u>code</u>)
departureAirport → Airports.code
destinationAirport → Airports.code
code → FlightCodes.code

a) Write a JSON schema corresponding to this database schema. Translate the relational schema as faithfully as possible (there is nothing you can do about the references, but can you have primary keys in JSON?).

b) Write a JSON document with the data in the table below, which validates with your schema.

Flight code	Airline	Dep.city	Dep.airport	Arr.city	Arr.airport
SK111	SAS	Gothenburg	GOT	Frankfurt	FRA
AF222	Air France	Paris	ORY	Malta	MLA
AB222	Air Berlin	Frankfurt	FRA	Munich	MUC
KM111	Air Malta	Munich	MUC	Malta	MLA

c) Write a Postgres SQL query that extracts the JSON data from the tables. The file ex5.2.sql contains some code to get you started.

d) Write a (Postgres) JSON Path that finds the flight code of all flights to Malta.

Hint: You can use the "additionalProperties" keyword to specify a schema for all properties of an object (except the ones mentioned in "properties").

Hint: Maybe it is easier to start with the document and then write the schema?

Solution:

a) Key/value pairs can be used to have primary keys (because there are no compound keys!).

Tiny example data (used to build schema):

```
{ "Airports": {"GOT": "Gothenburg"},
  "FlightCodes": {"SK111": "SAS"},
  "Flights": {"SK111" : {"dep" : "GOT", "dest": "FRA"}}
}
```

Schema:

```
{"type": "object",
 "properties": {
   "Airports": {
     "type":"object",
     "additionalProperties":{"type":"string"}
   },
   "FlightCodes": {
      "type":"object",
      "additionalProperties":{"type":"string"}
   },
   "Flights": {
      "type":"object",
      "additionalProperties":{
         "type":"object",
         "properties":{
                "dep":{"type":"string"},
```

```
                "dest":{"type":"string"}},
          "required": ["dep","dest"],
          "additionalProperties": false
       }
    }},
 "required": ["Airports","FlightCodes","Flights"],
 "additionalProperties": false
}
```

b) Complete data:

```
{ "Airports": {"GOT": "Gothenburg",
               "FRA": "Frankfurt",
               "ORY": "Paris",
               "MUC": "Munich",
               "MLA": "Malta"},
  "FlightCodes": {"SK111": "SAS",
                  "AF222": "Air France",
                  "AB222": "Air Berlin",
                  "KM111": "Air Malta"} ,
  "Flights": {"SK111" : {"dep" : "GOT", "dest": "FRA"},
              "AF222" : {"dep" : "ORY", "dest": "MLA"},
              "AB222" : {"dep" : "FRA", "dest": "MUC"},
              "KM111" : {"dep" : "MUC", "dest": "MLA"}}
}
```

c) Query to automatically export database:

```
-- Convert all three tables into one big JSON document
WITH
 Ap AS (SELECT json_object_agg(code, city) AS jsondata
        FROM Airports),

 Fc AS (SELECT json_object_agg(code, airlineName) AS jsondata
        FROM FlightCodes),

 F AS (SELECT
       json_object_agg(code,
               jsonb_build_object('dep', departureAirport,
                                  'dest', destinationAirport)
                     ) AS jsondata FROM Flights)

SELECT jsonb_pretty(
           jsonb_build_object('Airports', (SELECT jsondata FROM Ap),
                              'FlightCodes',(SELECT jsondata FROM Fc),
                              'Flights',(SELECT jsondata FROM F)));
```

d) `'strict $.Flights.keyvalue()?(@.value.dest == "MLA").key'` finds the flight code of all flights to Malta.

3 Applications

Below is some JSON data. It has been compiled by translating this schema in the most direct way possible:

Applicants (appNum, name)

Choices (applicant, code, choiceNum, score)
applicant → Applicants.appNum

```
{
  ‘‘Applicants’’: [
    {‘‘appNum’’: “a1”, ‘‘name’’: “Andersson”},
    {‘‘appNum’’: “a2”, ‘‘name’’: “Jonsson”},
    {‘‘appNum’’: “a3”, ‘‘name’’: “Larsson”}
  ],
  ‘‘Choices’’: [
    {‘‘applicant’’: “a1”, ‘‘code’’: “MPSOF”, ‘‘choiceNum’’: 1, ‘‘score’’: 750},
    {‘‘applicant’’: “a1”, ‘‘code’’: “MPALG”, ‘‘choiceNum’’: 2, ‘‘score’’: 750},
    {‘‘applicant’’: “a1”, ‘‘code’’: “MPCSN”, ‘‘choiceNum’’: 3, ‘‘score’’: 800},
    {‘‘applicant’’: “a2”, ‘‘code’’: “MPALG”, ‘‘choiceNum’’: 1, ‘‘score’’: 700},
    {‘‘applicant’’: “a3”, ‘‘code’’: “MPCSN”, ‘‘choiceNum’’: 1, ‘‘score’’: 850},
    {‘‘applicant’’: “a3”, ‘‘code’’: “MPALG”, ‘‘choiceNum’’: 2, ‘‘score’’: 850}
  ]
}
```

a) Can you rewrite the data into a more semi-structured format that uses the fact that there are no tables? Here are some suggestions:

- Avoid repeating applicant numbers (and the implicit references that exist in the data);
- Use key/value pairs instead of an array of rows;
- Maybe choice numbers are not needed?

b) Write a JSON schema for your modified data.

c) Write a (Postgres) JSONPath query on your modified data that finds all “choices” where choiceNum is 1 and score is greater than 800.

d) Write a (Postgres) JSONPath query on your modified data that finds the names of the applicants of the choices from above, that is, from all “choices” where choiceNum is 1 and score is greater than 800.

Solution:

a) Here we use the applicant id number as keys, and associate it with the applicant's name and a list of all his/her choices. We use array positions to represent choice numbers (indexed from 0 instead of 1).

```
{
  "a1": {"name": "Andersson",
         "choices": [ {"code":"MPSOF","score":750},
                      {"code":"MPALG","score":750},
                      {"code":"MPCSN","score":800}
                    ]
        },
  "a2": {"name": "Jonsson",
         "choices": [ {"code":"MPALG","score":700}]
        },
  "a3": {"name": "Larsson",
         "choices": [ {"code":"MPCSN","score":850},
                      {"code":"MPALG","score":850}
                    ]
        }
}
```

b)
```
{   "type": "object",
    "additionalProperties": {
        "type": "object",
        "properties": {
            "name" : {"type": "string"},
            "choices": {
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {
                        "code": {"type": "string"},
                        "score": {"type": "integer"}
                         },
                    "additionalProperties": false,
                    "required": ["code", "score"]
                }
            }
        },
        "required": ["choices", "name"],
        "additionalProperties": false
    }
}
```

c) Recall that application 1 has array index 0. Here are two possible ways to formulate the query.

```
'strict $.**.choices[0]?(@.score > 800)'
'strict $.**?(@.choices[0].score > 800).choices[0]'
```

d) Here we get the name of the applicant in question:

```
'strict $.**?(@.choices[0].score > 800).name'
```

4 Inventory of Computer Components

(From a previous exam.)

The following is part of an inventory of computer components at a webstore, divided into categories (such as GPUs, CPUs, ...). For each product, the information about manufacturer, model, full price and current discount for the upcoming sale is stored.

...

GPUs				
	Nvidia	3070GTX	11890kr	5%
	AMD	RX6800	13490kr	10%
CPUs				
	Intel	i7-12700K	4590kr	30%
	AMD	Ryzen 5600X	2739kr	12%
Memory				
	Corsair	LPX 32GB	1489kr	15%
	Kingston	Fury 64GB	2999kr	20%
	Kingston	Fury 128GB	4999kr	10%

...

a) Write a JSON document that encodes the data above. Make sure to keep the same structure as the data above.
To avoid writing a big document, it is enough that the inventory in your document fully contains just the first category (GPUs with all its products), with "..." to indicate the rest.

b) Write a JSON schema that describes your encoding of the data that matches the JSON document you provided in a). The schema needs to (at least) specify types of every JSON value in your encoding and the required properties of the objects.

c) Write a JSON path query that finds those products that cost 2500kr or less after the given discount has been applied.

Solution:

a) This is the whole data (as an array):

```
[   ...,
    {"category": "GPUs",
     "products": [
         {"manufacturer": "Nvidia",
          "model": "3070GTX",
          "price": 11890,
          "discount": 0.05
         },
         {"manufacturer": "AMD",
          "model": "RX6800",
          "price": 13490,
          "discount": 0.10
         }
     ]},
    {"category": "CPU",
     "products": [
         {"manufacturer": "Intel",
          "model": "i7-12700k",
          "price": 4590,
          "discount": 0.3
         },
         {"manufacturer": "AMD",
          "model": "Ryzen 5600X",
          "price": 2739,
          "discount": 0.12
         }
     ]},
    {"category": "Memory",
     "products": [
         {"manufacturer": "Corsair",
          "model": "LPX 32GB",
          "price": 4590,
          "discount": 0.15
         },
         {"manufacturer": "Kingston",
          "model": "Fury 64GB",
          "price": 2999,
          "discount": 0.20
         },
         {"manufacturer": "Kingston",
          "model": "Fury 128GB",
          "price": 4999,
          "discount": 0.10
         }
     ]},
     ...
```

```
]
```

but it's OK if you just fully write the first category and then '...' as stated in the question. Observe that your category needs to still be one of the element of an array!

```
[   ...,
    {"category": "GPUs",
     "products": [
         {"manufacturer": "Nvidia",
          "model": "3070GTX",
          "price": 11890,
          "discount": 0.05
         },
         {"manufacturer": "AMD",
          "model": "RX6800",
          "price": 13490,
          "discount": 0.10
         }
     ]},
    ...
  ]
```

b) The important bits here are that the outer structure is an array, and the discount and price are numbers and not strings.

```
{ "type": "array",
  "items": {"$ref": "#/definitions/categories"},
  "definitions": {
    "categories": {
      "type": "object",
      "properties": {
        "category": {"type": "string"},
        "products": {
          "type": "array",
          "items": {"$ref": "#/definitions/product"}
        }
      },
      "required": ["category","products"],
      "additionalProperties": false
    },
    "product": {
      "type": "object",
      "properties": {
        "manufacturer": {"type": "string"},
        "model": {"type": "string"},
```

```
        "price": {"type": "integer"},
        "discount": {"type": "number"}
      },
      "required": ["manufacturer","model","price","discount"],
      "additionalProperties": false
    }
  }
}
```

c) Here the solution depends on what the type of discount is.

If number as above then we have

```
'strict $.**?(@.price*(1-@.discount) <= 2500)'
```

If you model the discount as an integer, then you need to divide by 100 as in:

```
'strict $.**?(@.price*(1-(@.discount/100)) <= 2500)'
```