- Author: Petre Iordanescu
- Publisher: RENware Systems
- Edition: 2021 August 16
- Tags: software alghoritms
- Language: EN, partial RO

About brief series. These materials are derived from author training sessions as course notes for students. It is intended as presentation of basic topics regarding a concept. These materials are intended to present for a concept whst is good for, its main topics and what to deep research to find out and learn more about it.

Representing relational data as JSON

Introduction

Brief about data types as used in this article

Definition: A scalar data type represents a single, simple value type. No aggregation, no compounding, just simple as defined in the following.

Scalar data types

Mainly, we will use the following scalar data types:

- **number**, which ca be any number usable as is in mathematical operations, being it integer or float (real number)
- **string**, one or more characters, could be also an empty string which contains no character
- **boolean**, which can be true or false
- **null**, which represents a not yet defined value, nothing - please pay attention to not confuse with an empty string which in not a *null* value

Other scalar data types.

- **date**, which is a calendaristic date In fact, internally this is represented as a number (an increment from 1900) or as a string (in different formats, for example year/month/day)
- **boolean as number**, this is a boolean type where 0 means false and sny other number, true

Compounded data types.

The basic compounded data types are:

- **arrays**, these are *ordered* sequencers of other scalar values or other compounded values.
- **KV pairs** these are pairs of a value and a name for it. Is almost as define a variable, but the name of this variable is not hard coded, for example `"age": 31` establish the value 31 for key "age". This are often called *named value*. The value can be a scalar or other compounded value.
- **dictionaries**, these are collections of *KV pairs* under the same object named *dictionary*.

Notes and remarks.

- by ordered is understood that values can (must) be addressed using a known numerical index position
- in contrast, an unordered sequence can be addressed by a name (key), not by an numerical index
- ordered sequences are stored as contiguous blocks, so operations like previous and next obtained by decrementing or incrementing the index are perfectly possible
- on unordered sequences, arithmetic operations on keys are not guaranteed by default; only if we care to explicitly define the key as being numerical and, even in this case, operations like previous and next does not makes sense (by default)

Brief about relational data and & structures

Few words about physical storing data

This section intention is to give a very high level and abstract picture of how physical data is stored in relational database systems.

So data is stored in files and data blocks. Therefore there is a MUST to know the data length (in fact the maximum allowed length). This is an important thing and as consequence it is goof to know that physical space is allocated even it is not really used yet for useful information. This is true for almost all relational database systems. And where is possible to optimize the physical space, this will induce performance penalties (you often need to balance between performance and blocked space).

What is important to keep in mind, is that length of data ***must be known*** (at least un terms of maximums).

Columns

The columns represent *scalar types* but more precisely specified as length. Numbers must be known on how many bytes are represented (tinny as 1 byte, small int as 2 bytes, and so on). Strings aldo need to have a specific length and if you do not specify it, usual the system allocate 255 bytes.

Records

Records are collections of more columns. Records can be identified as a whole. The structure of records (what columns are part of) **is pre-established** and so almost hard coded. This is what is usually named **schemas**.

Tables

These are the top-most collection of data. They aggregates more records in one object.

There are some important aspects that need to be mentioned about tables:

- first is that the records in a table gas no specific order, more exactly, the order of records is not guaranteed *until you specify in clear at their retrieval one* (using ORDER BY clause).
- second is that a record need to be identified by using a *primary key*, which is your responsability to declare at *schema* level.
- some database servers use an internal ROWID or ROWNUM for their *internal purposes* to identify records, but this ia not guaranteed by any standard and neither the repeatability of such information is not guaranteed, neither between two queries in the same transaction scope.

That things established, the JSON model must accomodate them. It is also important ti mention that any JSON can accomodate the *referential integrities* assueed by RDBS. These are out if JSON scope and must be solved at code level.

Brief about JSON data & structures

JSON structures are nore appropriate than those used in programming languages. Also they offer almost the same versatility and flexibility.

Therefore, a JSON data structure operates with **KV pairs** having values of keys as:

- scalar: numbers, strings, booleans and null
- arrays of scalars or objects or combined (not uniform values)
- objects which are dictionaries of any complexity

It has the maximum flexibility letting to your needs both the content and definition of data structure. There is no predefined or default validation *schema*. There are many libraries usable for this purpose for the most known programming languages (Python, C, Java, and so on).

How to encapsulate a relational record in JSON

Array of records

The most agnostic model is representing records as an array:

```
[record, record, ... ]
```

and each record as an object (ie, dictionary) :

```
{
  "column name": value,
  "column name": value,
   ...
}
```

ROWID style

Another mode of representing records in JSON is by "imitating" the ROWID from relational:

```
{
  "ROWID_1": { // this is a record
    "column name": value,
    "column name": value,
   ...
  },
  "ROWID_2" : { // another record
    ...
  }, ...
}
```

Notes, comments

- In both formats, we used *string keys*. That's because the column name for sure will accommodate as string, agnostic for any RBDS
- be carefully and aware of different date types. This is in most cases a "jungle". If is possible use strings in universal formats, like `year/month/day hh:mm:ss:sss...` and for timezones use TZ qualifier or, if missing, the best assumption is for UTC. This will be in accordance with best practices.
- avoid comparison with `null` values. Best practice is to check before use with `is [not] null`. Therefore you will avoid potential fatal errors that will be experienced by end users.
- avoid equal comparison between date types. Be more "flexible" and make interval or greater or less than comparisons.
- avoid using timestamps as primary keys
- even if is a big temptation to use ROWID as primary key, DO NOT DO THAT. Do not mix the roles of each. Your primary key is the primary key. ROWID is for internal, local scope and limited context usage.
- when intend to send the JSON over Internet, do not forget to "*jsonify it*" or to properly encode and escape it of special characters (for example the / character which is used in URL route).

Recommended frameworks

- SQLalchemy as ORM
- Pydantic or Dataclass for complex class objects type validation
- FastAPI, Flask for REST like APIs
- PostGRE as RDBS able to manipulate JSON structures in string columns (and to use its keys as other normal columns)