JSON Lecture Notes

JSON is a very common format for storing and transmitting structured data *between* programs.

Additional Materials

- Sample SQL commands for this lecture
- URL for these notes: https://www.pg4e.com/lectures/06-JSON

Data Serialization 📖 🏠 ⏩

Each programming language has ways of representing the two core types of collections.

Language	Linear Structure	Key / Value Structure
Python	list() [1,2,3]	dict() {'a':1, 'b': 2}
JavaScript	Array [1,2,3]	Object {'a':1, 'b': 2}
PHP	Array array(1,2,3)	Array array('a' => 1, 'b' => 1)
Java	ArrayList	HashMap

In order to move structured data between applications, we need a "language independent" syntax to move the data. If for example, we want to send a dictionary from Python to PHP we would take the following steps:

1. Within Python, we would convert the dictionary to this "independent format" (serialization) and write it to a file.
2. Within PHP we would read the file and convert it to an associative array de-serialization).

Another term for serialization and deserialization is marshalling and unmarshalling.

A long time ago.... We used XML as this "format to move data structures between various languages":

```
<array>
    <entry>
       <key>a</key>
       <value>1</value>
    <entry>
    <entry>
       <key>b</key>
       <value>2</value>
    <entry>
</array>
```

XML (like HTML) is a good syntax to represent documents, but it is not a natural syntax to represent lists or dictionaries. We have been using XML as a way to represent structured data for interchange since the 1990's. Before that we had serialization formats like ASN.1 fsince the mid-1980s. And formats like Comma-Separated Values (CSV) work for linear structures but not so much for key value structures.

Around 2000, we started seeing the need to move structured data between code written in JavaScript in browsers (front-end) and code running on the servers (back-end). Initially the format of choice was XML resulting in a programming pattern called AJAX - Asynchronous JavaScript And XML. Many programming already had libraries to read and write XML syntax so it was an obvious place to start. And in the browser, XML looked a lot like HTML so it seemed to make sense there as well.

The problem was that the structures we used in programs (list and key/value) were pretty inelegant when expressed in XML, makeing the XML hard to read and a good bit of effort to convert.

JSON - JavaScript Object Notation ⏪ 📖 🏠 ⏩

Given the shortcomings of XML to represent linear and key/value structures, as more and more applications, started to transfer data between JavaScript on the browser and the databases on the back-end, Douglas Crockford noticed that the syntax for JavaScript constants might be a good serialization format. In particular, JavaScript already understood the format natively:

```
<script type="text/javascript">
who = {
    "name": "Chuck",
    "age": 29,
    "college": true,
    "offices" : [ "3350DMC", "3437NQ" ],
    "skills" : { "fortran": 10, "C": 10,
```

```
        "C++": 5, "python" : 7 }
};
console.log(who);
</script>
```

It turned out to be easier to add libraries to the back-end languages like Python, PHP, and Java to convert their data structures to JSON than to use XML to serialize data because the back-end languages were already good at XML. The reason was really because XML did a bad job of representing linear or key/value structures that are widely used across all languages.

To help advance adoption in an industry that was (at the time) obsessed with XML, Douglas Crockford wrote a simple specification for "JSON", and put it up at www.json.org and programmers started to use it in their software development.

In order to make parsing and generating JSON simpler, JSON required all of the keys of key value pairs be surrounded by double quotes.

For those familiar with Python, JSON looks almost exactly like nested Python list and dictionary constants. And while Python was not so popular in 2001, now almost 20 years later, with Python and JavaScript emerging as the most widely used languages, it makes reading JSON pretty natural for those skilled in either language.

JSON has quickly become the dominant way to store and transfer data structures between programs. JSON is sent across networks, stored on files, and stored in databases. As JavaScript became an emerging server language with the development of the NodeJS web server and JSON specific databases like MongoDB were developed, JSON is now used for all but a few data serialization use cases. For those document-oriented use cases like Microsoft Office XML formats, XML is still the superior solution.

Database systems like Oracle, SQLServer, PostgreSQL, and MySQL have been adding native JSON columns to suport document-style storage in traditional relational databases.

References

- Interview with Douglas Crockford

JSON in Python ⏪ 📖 🏠 ⏩

In this section we will do a quick introduction of the JSON library in Python.

Using JSON in Python is very simple because JSON maps perfectly onto lists and dictionaries.

The json.dumps() library takes a python object and serializes it into JSON.

https://www.pg4e.com/code/json1.py

```
import json

data = {}
data['name'] = 'Chuck'
data['phone'] = {}
data['phone']['type'] = 'intl';
data['phone']['number'] = '+1 734 303 4456';
data['email'] = {}
data['email']['hide'] = 'yes'

# Serialize
print(json.dumps(data, indent=4))
```

Produces the following output:

```
{
    "name": "Chuck",
    "phone": {
        "type": "intl",
        "number": "+1 734 303 4456"
    },
    "email": {
        "hide": "yes"
    }
}
```

The json.loads() takes a string containing valid JSON and deserializes it into a python dictionary or list as appropriate.

https://www.pg4e.com/code/json2.py

```
import json

data = '''
{
  "name" : "Chuck",
  "phone" : {
    "type" : "intl",
    "number" : "+1 734 303 4456"
   },
   "email" : {
     "hide" : "yes"
   }
}'''

info = json.loads(data)
print('Name:', info["name"])
print('Hide:', info["email"]["hide"])
```

This code executes as follows:

```
Name: Chuck
Hide: yes
```

This also works with Python lists as well:

- Serializing a list - https://www.pg4e.com/code/json3.py
- Deserializing a list - https://www.pg4e.com/code/json4.py

Before we move on, here is a simple example of de-serializing XML in Python similar to **json2.py** above:

https://www.pg4e.com/code/xml1.py

```
import xml.etree.ElementTree as ET

data = '''
<person>
  <name>Chuck</name>
  <phone type="intl">
    +1 734 303 4456
  </phone>
  <email hide="yes" />
</person>'''

tree = ET.fromstring(data)
print('Name:', tree.find('name').text)
print('Attr:', tree.find('email').get('hide'))
```

Because XML is a tree based approach (neither a list nor a dictionary) we have to use find **find()** function to query the tree, figure out its structure and *hand transform* the data tree into our lists and/or dictionaries. This is the impedance mismatch between the "shape" of XML and the "shape" of data structures inside programs that is mentioned by Douglas Crockford in his interview above.

Again, it is importaint to point out that XML is a better than JSON when representing things like hierarchical documents. Also XML is a bit more verbose and as such a bit more self-documenting as long as the XML tags have reasonable names.

Structured Data in PostgreSQL ⏪ 📖 🏠 ⏩

A key to understanding JSON support in PostgreSQL is that it has evolved. The great news is that since you are probably using PostgreSQL 10 or later - we will only talk about the historical perspective as history rather than having to use the old (somewhat clunky) support.

There are three supported column types in PostgreSQL that handle key/value or JSON data:

- HSTORE is column that can store keys and values. It frankly looks like a column that is a PHP Array / Python dictionary without support for nested data structures.

  ```
  pg4e=> SELECT 'a=>1,b=>2'::hstore;
         hstore
  --------------------
   "a"=>"1", "b"=>"2"
  (1 row)
  ```

  HSTORE stores key/value pairs efficiently and has good support for indexes to allow **WHERE** clauses to look *inside* the column efficiently. Indexes on HSTORE columns were easy to create and use (unlike the regular expression based indexes

we manually created in the gmane.py code).

- JSON (from PostgreSQL 9.3) is best thought of as a pre-release of JSONB. A **JSON** column was a glorified **TEXT** column with some really nifty built-in functions that kept application developers from "hacking up" their own JSON-like TEXT columns. Things like JSON operators and functions were nicely carried over into JSONB bring the best of JSON forward. This "layer of functions and indexes" on top of a TEXT column is a strategy that has been used by relational databases to quckly build and release JSON support to counter the move to NoSQL databases (more about that later).

- JSONB completely new column type that stores the parsed JSON densely to save space, make indexing more effective, and make query / retrieval efficient. The "B" stands for "better", but I like to think of it as "binary", ackmowledging that it is no longer a big TEXT column that happens to contain a JSON string.

In a sense, the JSONB support in PostgreSQL is a merger of the efficient storage and indexing of the HSTORE merged with the rich operator and function support of JSON.

But there are still some situations where HSTORE or JSON has an advantage over JSONB - you can research that question online. But for most aplications, just use JSONB even for simple key/value applications that might work well with HSTORE. It is less to remember and there will be a lot of investment and performance tuning that goes into JSONB in future versions of PostgreSQL as it competes with all the NoSQL databases.

References

- When to use unstructured datatypes in Postgres–Hstore vs. JSON vs. JSONB (Blog Post)
- How do I make Postgres extension like hstore available to non superuser
- Why we Moved From NoSQL MongoDB to PostgreSQL (Blog Post)
- PostgreSQL, the NoSQL Database (Linux Journal)

JSONB in PostgreSQL ⏪ 📖 🏠 ⏩

Now we *finally* get to talk about JSONB support in PostgreSQL. Like many of things with PostgreSQL, the lead up / background is more complex to understand than the support within PostgreSQL.

In this section, we will be going back to our music data except we will now be using JSON data. If you are interested, you can see the Python code which is used to convert the original XML data is converted to JSON format.

```
{"name": "Another One Bites The Dust", "artist": "Queen", "album": "Greatest Hits", "count": 55, "rating": 100, "length": 217103}
{"name": "Beauty School Dropout", "artist": "Various", "album": "Grease", "count": 48, "rating": 100, "length": 239960}
{"name": "Circles", "artist": "Bryan Lee", "album": "Blues Is", "count": 54, "rating": 60, "length": 355369}
```

Separately, we will go over a detailed walkthrough of SQL statements using JSONB, so we will just show some of the highlights here.

We will create a table and import the above JSON file into the table as follows:

```
CREATE TABLE IF NOT EXISTS jtrack (id SERIAL, body JSONB);

\copy jtrack (body) FROM 'library.jstxt' WITH CSV QUOTE E'\x01' DELIMITER E'\x02';
```

The **\copy** command above is somewhat inelegant but it got our data in with a single command. In the next section of these notes, we will insert our JSON data using Python which gives us a lot more flexibility.

When using JSONB it is important to know the types of data and use the cast (::) operator where appropriate. You can extract field data from the JSON using queries that use the "retrieve field and convert from jsonb to text" operator (->>).

```
SELECT (body->>'count')::int FROM jtrack WHERE body->>'name' = 'Summer Nights';
```

You can query JSONB fields by comparing them to other JSONB documents or document fragments using the contains (@>) operator.

```
SELECT (body->>'count')::int FROM jtrack WHERE body @> '{"name": "Summer Nights"}';
```

You can check to see if a JSONB document contains a key:

```
SELECT COUNT(*) FROM jtrack WHERE body ? 'favorite';
```

You can use JSONB expressions most anywhere you can use a column in your SQL, making sure to cast the results where appropriate.

```
SELECT body->>'name' AS name FROM jtrack ORDER BY (body->>'count')::int DESC;
```

Indexes

Part of the benefit of using JSONB is the way you can easily add indexes to the whole column or portions of the column using BTREE, HASH, Gin and other types of PostgresSQL indexes:

```
CREATE INDEX jtrack_btree ON jtrack USING BTREE ((body->>'name'));
CREATE INDEX jtrack_gin ON jtrack USING gin (body);
CREATE INDEX jtrack_gin_path_ops ON jtrack USING gin (body jsonb_path_ops);
```

We will look at the kinds of **WHERE** clauses that make use of the various indexes.

References

- JSON Functions and Operators
- PostgreSQL internals: JSONB type and its indexes (Blog Post)

Sample Code: Loading JSON from an API ⏪ 📖 🏠

In this sample code walkthrough, we will use the Star Wars API to spider a JSON data source, and pull it into a database and then work with the data using SQL.

Download these files

- https://www.pg4e.com/code/swapi.py
- https://www.pg4e.com/code/myutils.py

Make sure the **hidden.py** is set up and has your credentials.

The database for this program is more complex. In addition to a column for the JSON data we have fields to help make our data spidering process "smart" so it only retrieves a particular document once.

```
CREATE TABLE IF NOT EXISTS swapi (
  id SERIAL, body JSONB,
  url VARCHAR(2048) UNIQUE, status INTEGER,
  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
  updated_at TIMESTAMPTZ
);
```

This application works somewhat like Google search because each time it retrieves a JSON document, it scans the document for urls to other documents available from the API, and adds these URLs to the database in a state of "to be retrieved". The program runs over and over, reading an unread URL, parsing the data, inserting it into the database and checking for new URLs in the document and iterating.

If you run the program long enough, it finds all of the documents available in this API (turns out to be just over 200) and stops. As you are only retrieving a few hundred documents, you probablly will not run into the rate limit of this API. If you do you will have to wait a bit and restart the program.

This is an example of the first run with an empty database:

```
python3 swapi.py

INSERT INTO swapi (url) VALUES ( 'https://swapi.py4e.com/api/films/1/' )
INSERT INTO swapi (url) VALUES ( 'https://swapi.py4e.com/api/species/1/' )
INSERT INTO swapi (url) VALUES ( 'https://swapi.py4e.com/api/people/1/' )
Total=3 todo=3 good=0 error=0
How many documents:10
200 2201 https://swapi.py4e.com/api/films/1/ 2
200 1883 https://swapi.py4e.com/api/species/1/ 34
200 702 https://swapi.py4e.com/api/people/1/ 39
200 505 https://swapi.py4e.com/api/species/5/ 41
200 661 https://swapi.py4e.com/api/species/3/ 40
200 750 https://swapi.py4e.com/api/species/2/ 39
200 473 https://swapi.py4e.com/api/species/4/ 38
200 478 https://swapi.py4e.com/api/vehicles/4/ 37
200 433 https://swapi.py4e.com/api/vehicles/6/ 36
200 443 https://swapi.py4e.com/api/vehicles/7/ 35
How many documents:

Loaded 10 documents, 8529 characters
Total=45 todo=35 good=10 error=0
Closing database connection...
```

At the end of this run, it has retrieved ten documents and has 35 documents on the to-do list.

Any time during or after the run, you can use **psql** in another window and check the progress of the job using commands like:

```
-- How many urls total?
SELECT COUNT(url) FROM swapi;

-- What are the unretrieved URLs?
SELECT url FROM swapi WHERE status != 200;
```

Since it is a "spider" and restartable, you can run the program again and it will pick up where it left off and work on retrieving documents on the "to do list".

```
python3 swapi.py

Total=45 todo=35 good=10 error=0
How many documents:5
200 524 https://swapi.py4e.com/api/vehicles/8/ 34
200 560 https://swapi.py4e.com/api/starships/2/ 33
200 574 https://swapi.py4e.com/api/starships/3/ 32
200 533 https://swapi.py4e.com/api/starships/5/ 31
200 581 https://swapi.py4e.com/api/starships/9/ 30
How many documents:

Loaded 5 documents, 2772 characters
Total=45 todo=30 good=15 error=0
Closing database connection...
```

When you have all 200+ documents loaded, when you run the spider it will just shut down because it has nothing on its to-do list.

At that point, we start playing with our retrieved JSON using SQL.
