Python in Healthcare

ETL Processing of Nested Data

May 11, 2017

Workshop:
ETL Processing of Nested Data

ETL JSON Processing - Agenda

- Bio
- Problem Description
- Selected Python Techniques
- Transforming Nested Data
- Optimization Techniques
- Pitfalls
- Q & A

Hi, I’m Bijan!

- From Canada
- Started writing code at age 7

- B. Sc. in Software Engineering and Human Biology 2000
- Entered workforce as Software Developer since 1999
- MBA 2007
- Worked in FX/Derivative Trading, Gaming, D/R, and Network Apps
- Javascript, C/C++, Java, C#, perl, and ruby
- Worked on most modern O/S platforms

- Python developer since 2012
- Senior Developer with WebApps team @Clover Health since 2016

Source Code

Available here:

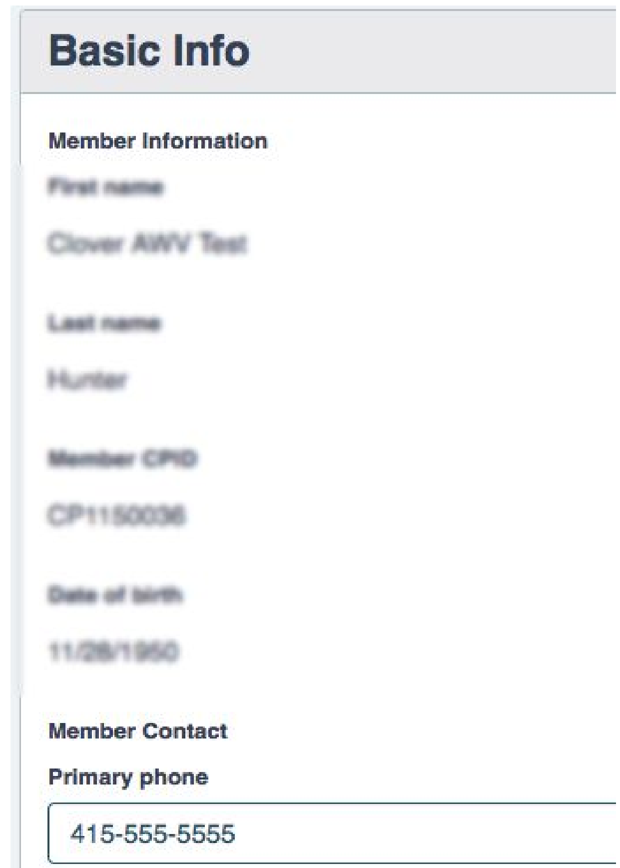
https://github.com/CloverHealth/pycon2017

Problem Description

Data collected via forms uses nested JSON in two shapes:

“Schema” describes Questions

```
{
  "label": "Basic Info",
  "slug": "basic_info",
  "widget": "panel",
  "nodeType": "section",
  "children": [
    {
      "slug": "primary_phone",
      "label": "Primary Phone",
      "nodeType":"question",
      "answerType": "text",
      "widget": "phone_number_control",
    },
   . . .
]
```



Problem Description

Data collected via forms uses nested JSON in two shapes:

“Submission” captures Answers

```
{
  "basic_info": {
    "member_info": {

      "vitals": {
        "name": "McTestface",
        "age": 42,
        "favorite_color": "blue",
        "bmi": 23
      }

    }

}
```

Problem Description - Nesting

```
{
  "slug": "basic_info",
  . . .
  "children": [
    {
      "slug": "member_info",
       . . .
       "children": [
           {
             "slug": "contact_info",
             ...
              "children": [
                 {
                     "slug": "phone",
                      ...
                 },
                . . .
```

Problem Description - Sample Output

schema_path (key)	***value***
basic_info.member_info.vitals.**name**	“McTestface”
basic_info.member_info.vitals.**age**	“42”
basic_info.member_info.vitals.**favorite_color**	“blue”
basic_info.member_info.vitals.**bmi**	“24”
. . .	. . .
other_form.new_section.info.**bmi**	“26”

Problem Description - ETL Process

"ETL" = **Extract** + **Transform** + **Load**

- Extraction — *Reading* source data
- Transform — *Converting or computing* output data
- Load — *Inserting* output data

Our ETL application will convert our data to a **flat** key/value table.

Why?

- Nested JSON is cumbersome to query in Postgres
- Different form schemas have different nested structures
- Single data format for easier "downstream" processing
  - e.g. metrics, reports, pivot views, etc.

Python Technologies Used

Core python

Generators	Chaining flow of data Conserve memory
`functools`	Partial functions for encapsulation Caching
`more_itertools`	Batching / chunking

Data Access

`SQLAlchemy`	Object Relational Mapper (ORM)
`psycopg2`	Postgres database driver
`testing.postgresql`	Manage disposable/reuseable test database

Python Generators

Generators are a powerful but simple technique for

- Concisely creating iterators
- Deferred evaluation
- Incremental evaluation
- Chaining operations

How?

- Use **`yield`** keyword to “generate” a value
- Returns an *iterable* object. Not a list!
- Evaluate at your own leisure
- Save memory.
- Easy to “chain” generators

Simple example: **`squares()`** generator function

Python Generators

Simple examples in `core_techniques.ipynb`

- `squares()` generator function
- Chaining generators (`fun_A()`, `fun_B()` and `fun_C()`)

Partial Function Application

Partial functions can be created in python using `functools.partial()`:

- "Freezes" some portion of a function's args and/or kwargs
- Creates a new callable object
- Can still be called as a function
- Simplifies signature during execution

Back to examples in **`core_techniques.ipynb`**

Partial Function Application - Encapsulation

Encapsulation = “Construct that bundles data with methods”

You can do this in python using:

● Classes	Data attributes in `__init__()`
● Closures / Inner methods	Variables from outer scope
● Partial (function) application	“Frozen” args/kwargs for a function

These are all *valid* approaches.

Workshop code uses partial function application because:

- Quickly kwargs parameters from JSON config
- This *particular* application doesn’t need class hierarchies
- Each of E, T and L only performs a *single operation*

Workshop Application - Core Design

- `app.etl.extractors` Read routines (extraction)
- `app.etl.transformers` JSON Transformation code
- `app.etl.loaders` Result insertion routines (loading)

- `app.processor` Main ETL flow in just 3 lines of code
  - Uses python generators to all 3 steps
  - Uses partial functions to encapsulate ETL parameters

See **`core_techniques.ipynb`** for simplified example

Transforming Nested Data

- Use a “mapping function” approach
  - Take iterable or generator
  - Apply functions to transform the data
  - Generate transformed results

- Known python built-in examples include:
  - `map()` - applies a function to an iterable
  - `sorted()` - applies a function to extract an ordering “key”

- For nested JSON structures, we need to:
  - Traverse the tree
  - Filter only required data from each tree node
  - Filter only required children

Transforming Nested Data

Workshop introduces app.etl.transformers.`map_nested()`

- INPUTS
  - python dict -- loaded JSON data
  - gen function to transform items at each node
  - gen function extract children to recurse into
- OUTPUTS:
  - transformed items (as a gen function)

- CORE ALGORITHM:
  - Pre-order DFS (depth-first search)
  - Current path (node slugs) are tracked using an array argument
  - Just 5 lines of code --- WAAAAAT?
    - Complexity resides in your INPUT generator functions

Transforming Nested Data - Quick App Demo

- Input Scenario: “`demo`”
  - 2 form schemas
  - 50 users
  - 50 submissions
- Processor setup: “`naive-single`”
  - Iterate on `Session.query()`
  - Call `Session.add()` for each response event
- Run:
  - `python main.py` **`generate`** `demo`
  - `python main.py` **`process`** `demo` **`naive-single`**

See `README.md` for detailed instructions

Transforming Nested Data - Performance

Even with generators, ETL processing of large amounts of data can incur performance problems

- ETL everything in memory?
  - … but memory is limited

- ETL one row at a time?
  - … will be very slow due to lots of I/O

Performance Optimization - Batching

Find a balance using **batches** (or “chunks”)

SQLAlchemy provides:

- `Query.yield_per()` for extraction using database CURSORs
- `Session.bulk_save_objects()` and `Session.bulk_insert_mappings()` for load insertion

Performance Optimization - Batching

The “T” in ETL (the Transform) is <u>not</u> necessarily 1 to 1

- 1 row SELECTed will generally <u>not</u> lead to 1 row to INSERTed

In our workshop:

- *One* JSON structure will lead to *multiple* key-value pairs
- Different schemas will produce different number of key-value pairs

Do we need to write the “chunking” logic ourselves?

Performance Optimization - Batching

`more_itertools` library comes to the rescue!

- `chunked()` method can take *any* input iterable
- Produces controlled chunks
- No complex logic needed

Loader can easily apply this to transformed output generator to reduce INSERTs

See **`core_techniques.ipynb`** for simplified example

`chunked()` is just 1 line of code

… it uses a partial function!!

Performance Optimization - Caching

- For each form schema, the transformer computes a “path map”
  - `basic_info.member_info.name -> text`
  - `basic_info.member_info.age -> number`

- Optimize via caching by using `functools.lru_cache()`

- But read the python docs carefully!
  - “the [...] arguments to the function must be hashable.”

- The SQLAlchemy “`session`” an opaque object.
  - Hashing would include internal state which changes
  - (loaded objects, cursors, etc.)

Performance Optimization - Caching

- No problem… use partial functions again!

- Create a custom function using [`functools.partial()`]
  - Freeze the "`session`" argument (it's an opaque reference)
  - `form_id` is then the only (hashable) parameter

- Wrap the resulting partial function with [`functools.lru_cache()`]

- Client code just calls `get_node_path_map(form_id)`

Performance Pitfalls - SQLAlchemy ORM

SQLAlchemy ORM keeps code concise....

- Great for for general purpose in transactional apps
- e.g. Web sites, REST APIs, GraphQL
- Anything “CRUD”

… but gets tricky when dealing large datasets in OLAP

- e.g. Data Science, Analytics, Batch processing
- Default lazy loading leads to more SELECTs instead of joins (I/O)
- Transforming python objects to model instances (CPU)

Performance Pitfalls - SQLAlchemy ORM

For analytics, please consider any of these options:

- Explicitly specifying join load types if you must use the ORM
  - see **`join_queries`**`.ipynb` included

- SQLAlchemy Core to compose exact queries
  - Allows more precise queries
  - Be careful with readability!

- Using raw SQL
  - Best query precision (especially for advanced Postgres)
  - Often more readable
  - Via SQLAlchemy's [Session.execute()]()

Bonus! - Performance Analysis

This workshop code include performance test content!

- Timing and Memory Profiling
- Demonstrate tradeoffs between choices
- Repeatable due to “scenarios” template database

Run “`jupyter notebook`” to get detailed explanations on:

- **`etl_analysis`**`.ipynb` Performance tradeoffs
- **`join_queries`**`.ipynb` ORM join options for extraction

Please: Ask Bijan questions after the presentation!

Bonus! - Performance Analysis Tools

Profiling and Investigation

`cProfile` (core python)	Time profiling
`memory_profiler`	Memory profiling

Presentation and Visualization

`jupyter`	Detailed write-up on analysis
`gprof2dot`	Visualize timing profiling (from `cProfile`)
`matplotlib`	Visualize memory profiling (from `memory_profiler`)
`sqlparse`	Pretty-printing raw SQL in logs (*)

(*) NOTE: In healthcare, SQL debug poses risk of leaking protected information.

This workshop provides SQL debug logging, but all data is fake!

Bonus! - Performance Analysis Commands

You can analyze what is happening with the application:

- *Time* profiling

```
python -m cProfile -o timing.stats main.py process demo naive-single
bash visualize_pstats.sh timing.stats
```

- *Memory* profiling

```
mprof run python main.py process demo naive-single
mprof plot
```

- SQL debug logging

```
python main.py --debug-sql process demo naive-single > sql.log
less sql.log
```

Q & A

Credits

This workshop would not have been possible without prior work from these folks at Clover:

- James Bennett
- Joey Leingang
- David Flerlage
- Kathy Lass
- Diego Argueta
- Lavinia Karl
- Paul Minton

Thank you & contact information

Bijan Vakili

Senior Developer, Clover Health

bijan.vakili@cloverhealth.com

Appendix

Testing Approach

Unit Tests

- Run all tests within DB transactions that are always rolled back
  - Use **`Session.flush()`** instead of `Session.commit()`
- Extend pytest CLI options to allow reuse a preset database
  - Uses the **`base_dir`** option in `testing.postgres`
  - Preset with schemas and no data
- `freezegun:`Artificially “Freeze” the system clock in unit tests

Performance Tests

- Reuse template database with preset datasets of any size
  - Uses the **`copy_data_form`** option in `testing.postgres`
  - Preset with schemas, forms and responses
  - No output events

Decision to use JSONB

- Faster reads for this workshop - no reparsing necessary
  - (but slower writes than JSON)
- No need to preserve semantically-insignificant whitespace between tokens
- No need to preserve key order
- Can support more Postgres operators and indexing

Pitfalls - SQLAlchemy, Postgres and timestamps

Be aware of how you use timestamps

- Always Include timezone
  - Stick with UTC wherever possible
- Watch out for time truncation by `psycopg2` driver

Workshop Application - Usage

Execution: `python main.py …`

`generate`	Create a template dataset (“scenario”)
`process`	Run the ETL transform using a scenario
`psql`	Connect to the resulting database and review the results
`clean`	Cleanup

Configuration: Edit `conf/…`

`schemas/*.json`	Sample JSON form schemas
`perfdata.conf.json`	Describe your scenarios
`processors.conf.json`	Tune your processor

See `README.md` for instructions

Transforming Nested Data

INPUT: Submissions (nested JSON responses)

OUTPUT: Response events `(schema_path, value, tag)`

`app.etl.`**`transformers`** module uses:

- Recursion to traverse nested subtrees
- Generators to `yield` events

Clover Platform - Common Elements

Infrastructure:

- Aptible for “Compliance as a Service”...
  - *PaaS Docker containers with healthcare compliance (HIPAA certified)*
  - Postgres DBs
  - SSL certificates
- ...on Amazon AWS
  - EC2, S3, Route 53, . . .

Critical Integrations

- Bug Capture/Monitoring - Sentry (`raven`)
- Monitoring - New Relic + Pager Duty (`pygerduty`)

Pitfalls - SQL logging

SQL debug logging is a great technique for finding root causes...

… but is discouraged in healthcare.

Sensitive member/patient information can be leaked in logs because of:

- INSERT parameters
- SELECT results

- “Data cleaning” tools exists… but customization effort is required

- This workshop’s code suppresses parameters and results in its logs
  - `app.log`
  - `app.util.sqldebug`