JSON Schema ➢ Database

better.engineering/jsonschema2db

A simple utility to convert JSON Schemas into relational tables in Postgres/Redshift.

Also see the Github page for source code and discussion!

Installation

The easiest way to install is from PyPI:

```
pip install jsonschema2db
```

Quick overview

Let's say you have the JSON schema test_schema.json:

```
{
    "$schema": "http://json-schema.org/schema#",
    "title": "Fact schema",
    "type": "object",
    "definitions": {
        "basicAddress": {
            "type": "object",
            "comment": "This is an address",
            "properties": {
                "City": {
                    "type": "string",
                    "comment": "This is a city"
                },
                "State": {
                    "type": "string",
                    "minLength": 2,
                    "maxLength": 2
                },
                "Street": {
                    "type": "string"
                },
                "ZipCode": {
                    "type": "string"
                }
            }
        },
        "address": {
            "allOf": [
                {
                    "$ref": "#/definitions/basicAddress"
                },
                {
                    "type": "object",
                    "properties": {
                        "Latitude": {
                            "type": "number",
                            "minimum": -90,
                            "maximum": 90
                        },
                        "Longitude": {
                            "type": "number",
                            "minimum": -180,
                            "maximum": 180
                        }
                    }
                }
            ]
        },
        "unum": {
            "type": "number",
            "minimum": 0
        }
    },
    "properties": {
        "Loan": {
            "type": "object",
            "description": "Loan information",
            "properties": {
                "Amount": {
```

```
                    "type": "number"
                },
                "SomeSuperLongColumnNameThatDoesntFitInPostgresUnfortunately": {
                    "type": "number"
                },
                "AbbreviateThisReallyLongColumn": {
                    "type": "number"
                }
            }
        },
        "SubjectProperty": {
            "type": "object",
            "properties": {
                "Acreage": {
                    "$ref": "#/definitions/unum"
                },
                "Address": {
                    "$ref": "#/definitions/address"
                }
            }
        },
        "RealEstateOwned": {
            "type": "object",
            "patternProperties": {
                "[0-9]+": {
                    "type": "object",
                    "properties": {
                        "Address": {
                            "$ref": "#/definitions/basicAddress"
                        },
                        "RentalIncome": {
                            "type": "number"
                        }
                    }
                }
            }
        }
    }
}
```

This looks a bit complex, but basically:

1. There's a shared definition for an *basicAddress* definition that has the normal address fields: state, zip, etc
2. There's a definition *address* that extends *basicAddress* and adds latitude and longitude
3. Each "loan" (we are in the mortgage industry) tracks a loan amount, a bunch of info (including address) for the subject property (the property that the loan is against), and a list of other properties owned by the borrower (each of which has an address and a rental income)

jsonschema2db creates the following tables automatically:

```
create table "schm"."root" (
   id serial primary key,
   "loan_file_id" int not null,
   "prefix" text not null,
   "loan__amount" float,
   "subject_property__acreage" float,
   "subject_property__address__latitude" float,
   "subject_property__address__longitude" float,
   "subject_property__address_id" integer,
   unique ("loan_file_id", "prefix")
)

create table "schm"."basic_address" (
   id serial primary key,
   "loan_file_id" int not null,
   "prefix" text not null,
   "city" text,
   "root_id" integer,
   "state" text,
   "street" text,
   "zip_code" text,
   unique ("loan_file_id", "prefix")
)

create table "schm"."real_estate_owned" (
   id serial primary key,
   "loan_file_id" int not null,
   "prefix" text not null,
   "address_id" integer,
   "rental_income" float,
   "root_id" integer,
   unique ("loan_file_id", "prefix")
)
```

As you can see, we end up with three tables, each containing a flat structure of scalar values, with correct types. jsonschema2db also converts camel case into snake case since that's the Postgres convention. Unfortunately, Postgres limits column names to 63 characters (127 in Redshift). If you have longer column names, provide a list of abbreviations using the *abbreviations* parameter to the constructor.

jsonschema2db also handles inserts into these tables by transforming them into a flattened form. On top of that, a number of foreign keys will be created and links between the tables.

The rule for when to create a separate table is that either:

1. It's a shared definition that is an object (with links from the parent to the child)
2. Any object with *patternProperties* will have its children in a separate table (with links back to the parent, if the link is unique)

Creating tables

The first step is to instantiate a `jsonschema2db.JSONSchemaToPostgres` object (or the corresponding `jsonschema2db.JSONSchemaToRedshift` and create the tables using `jsonschema2db.JSONSchemaToPostgres.create_tables()`:

```
schema = json.load(open('test/test_schema.json'))
translator = JSONSchemaToPostgres(
    schema,
    postgres_schema='schm',
    item_col_name='loan_file_id',
    item_col_type='string',
    abbreviations={
        'AbbreviateThisReallyLongColumn': 'AbbTRLC',
    }
)

con = psycopg2.connect('host=localhost dbname=jsonschema2db-test')
translator.create_tables(con)
```

Inserting data

Now, let's insert some data into the tables:

```
translator.insert_items(con, [
    ('loan_file_abc123', {
        'Loan': {'Amount': 500000},
        'SubjectProperty': {'Address': {'City': 'New York', 'ZipCode': '12345',
'Latitude': 43}, 'Acreage': 42},
        'RealEstateOwned': {'1': {'Address': {'City': 'Brooklyn', 'ZipCode':
'65432'}, 'RentalIncome': 1000},
                            '2': {'Address': {'City': 'Queens', 'ZipCode':
'54321'}}},
    })
])
```

This will create the following rows:

```
jsonschema2db-test=# select * from schm.root;
-[ RECORD 1 ]------------------------+-----------
id                                   | 1
loan_file_id                         | loan_file_abc123
prefix                               |
loan__amount                         | 500000
subject_property__acreage            | 42
subject_property__address__latitude  | 43
subject_property__address__longitude |
subject_property__address_id         | 1

jsonschema2db-test=# select * from schm.basic_address;
-[ RECORD 1 ]+--------------------------
id           | 2
loan_file_id | 1000000000
prefix       | /RealEstateOwned/1/Address
city         | Brooklyn
root_id      | 1
state        |
street       |
zip_code     | 65432
-[ RECORD 2 ]+--------------------------
id           | 1
loan_file_id | 1000000000
prefix       | /SubjectProperty/Address
city         | New York
root_id      | 1
state        |
street       |
zip_code     | 12345

jsonschema2db-test=# select * from schm.real_estate_owned;
-[ RECORD 1 ]-+------------------
id            | 1
loan_file_id  | 1000000000
prefix        | /RealEstateOwned/1
address_id    | 2
rental_income | 1000
root_id       | 1
```

Post-insertion

After you're done inserting, you generally want to run `jsonschema2db.JSONSchemaToPostgres.create_links()` and `jsonschema2db.JSONSchemaToPostgres.analyze()`. This will add foreign keys and also analyze the table for better performance.

Full API documentation

class `jsonschema2db.` `JSONSchemaToDatabase` **(*schema*, *database_flavor*, *postgres_schema=None*, *debug=False*, *item_col_name='item_id'*, *item_col_type='integer'*, *prefix_col_name='prefix'*, *abbreviations={}*, *extra_columns=[]*, *root_table='root'*, *s3_client=None*, *s3_bucket=None*, *s3_prefix='jsonschema2db'*, *s3_iam_arn=None*)[source]**
JSONSchemaToDatabase is the mother class for everything

Parameters

- **schema** – The JSON schema, as a native Python dict
- **database_flavor** – Either “postgres” or “redshift”
- **postgres_schema** – (optional) A string denoting a postgres schema (namespace) under which all tables will be created
- **debug** – (optional) Set this to True if you want all queries to be printed to stderr
- **item_col_name** – (optional) The name of the main object key (default is ‘item_id’)
- **item_col_type** – (optional) Type of the main object key (uses the type identifiers from JSON Schema). Default is ‘integer’
- **prefix_col_name** – (optional) Postgres column name identifying the subpaths in the object (default is ‘prefix’)
- **abbreviations** – (optional) A string to string mapping containing replacements applied to each part of the path
- **extra_columns** – (optional) A list of pairs representing extra columns in the root table. The format is (‘column_name’, ‘type’)
- **root_table** – (optional) Name of the root table
- **s3_client** – (optional, Redshift only) A boto3 client object used for copying data through S3 (if not provided then it will use INSERT statements, which can be very slow)
- **s3_bucket** – (optional, Redshift only) Required with s3_client
- **s3_prefix** – (optional, Redshift only) Optional subdirectory within the S3 bucket
- **s3_iam_arn** – (optional, Redshift only) Extra IAM argument

Typically you want to instantiate a *JSONSchemaToPostgres* object, and run `create_tables()` to create all the tables. After that, insert all data using `insert_items()`. Once you’re done inserting, run `create_links()` to populate all references properly and add foreign keys between tables. Optionally you can run `analyze()` finally which optimizes the tables.

`analyze` **(*con*)[source]**
Runs *analyze* on each table. This improves performance.

See the Postgres documentation for Analyze

`create_links` **(*con*)[source]**
Adds foreign keys between tables.

`create_tables` **(*con*)[source]**
Creates tables

Parameters
con – psycopg2 connection object

`insert_items` **(*con*, *items*, *extra_items={}*, *mutate=True*, *count=False*)**
[source]
Inserts data into database.

Parameters
- **con** – psycopg2 connection object
- **items** – is an iterable of tuples *(item id, values)* where *values* is either:
- A nested dict conforming to the JSON spec
- A list (or iterator) of pairs where the first item in the pair is a tuple specifying the path, and the second value in the pair is the value.

Parameters
- **extra_items** – A dictionary containing values for extra columns, where key is an extra column name.
- **mutate** – If this is set to *False*, nothing is actually inserted. This might be useful if you just want to validate data.
- **count** – if set to *True*, it will count some things. Defaults to *False*.

Updates *self.failure_count*, a dict counting the number of failures for paths (keys are tuples, values are integers).

This function has an optimized strategy for Redshift, where it writes the data to temporary files, copies those to S3, and uses the *COPY* command to ingest the data into Redshift. However this strategy is only used if the *s3_client* is provided to the constructor. Otherwise, it will fall back to the Postgres-based method of running batched insertions.

Note that the Postgres-based insertion builds up huge intermediary datastructures, so it will take a lot more memory.

class `jsonschema2db.JSONSchemaToPostgres` **(**args*, ***kwargs*)[source]**
Shorthand for JSONSchemaToDatabase(..., database_flavor='postgres')

class `jsonschema2db.JSONSchemaToRedshift` **(**args*, ***kwargs*)[source]**
Shorthand for JSONSchemaToDatabase(..., database_flavor='redshift')