Three Steps from SQL to a Document Database

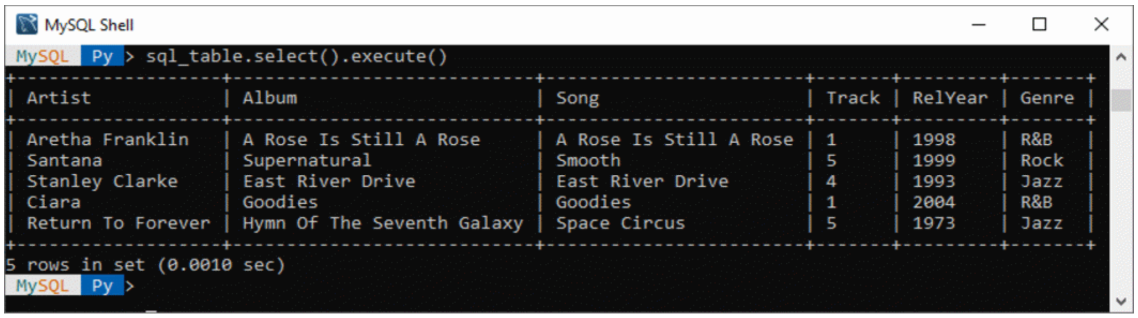
How I used a Python API to migrate my Music Library from SQL to a document database

Overview

In this article, I will show you how I used a Python application programming interface (API) to migrate my music library from a SQL relational database to a NoSQL document database. Using the Python X DevAPI in the MySQL Shell application, I will highlight some basics about document databases, the Python methods that I used, and the database tool that enables migration. Readers who should get the most out of this article are those that have some basic familiarity with the structured query language (SQL) and with the Python programming language.

Why migrate from SQL to document?

I have my existing personal music library in a SQL relational database containing music metadata — artist, song title, album title, track number, genre, release year — that I want to migrate to a document database. For the purpose of this article, I will use a few examples from my library.



```
MySQL  Py > sql_table.select().execute()
+-------------------+---------------------------+---------------------+-------+---------+-------+
| Artist            | Album                     | Song                | Track | RelYear | Genre |
+-------------------+---------------------------+---------------------+-------+---------+-------+
| Aretha Franklin   | A Rose Is Still A Rose    | A Rose Is Still A Rose | 1  | 1998    | R&B   |
| Santana           | Supernatural              | Smooth              | 5     | 1999    | Rock  |
| Stanley Clarke    | East River Drive          | East River Drive    | 4     | 1993    | Jazz  |
| Ciara             | Goodies                   | Goodies             | 1     | 2004    | R&B   |
| Return To Forever | Hymn Of The Seventh Galaxy | Space Circus       | 5     | 1973    | Jazz  |
+-------------------+---------------------------+---------------------+-------+---------+-------+
5 rows in set (0.0010 sec)
MySQL  Py >
```

Figure 1: Sample table of songs from my SQL library

SQL relational databases have been dominant for decades, making up 60% of the database market in of 2019 according to a ScaleGrid Database Trends report. In recent years, use of document databases has increased, largely driven by the requirements of big data. One of the criticisms of relational databases is that their schema is rigid. All data fields must be defined in advance with identical fields in every row in a table making it difficult to make schema changes later.

By contrast, document store databases, sometimes referred to as NoSQL, do not have a fixed schema. Document databases do not require each document to have the same fields, but they can. In fact, it is possible to have different fields in each document throughout the database. That flexibility is one of the key advantages of a document store over a relational store. It is the reason I decided to migrate because it allows me to easily add new metadata to my music library. That could include metadata such as artist background information, song credits, and/or other miscellaneous metadata that may not be immediately available.

What is a JSON document?

In many document store systems, documents are JavaScript object notation (JSON) objects, or JSON-like objects. JSON is becoming increasingly popular as a standard for data interchange and storage and is beginning to replace the extensible markup language (XML) as a dominant data exchange format, particularly for music metadata. JSON documents are lightweight, language-independent, and human readable. In short, JSON documents are elegant in their simplicity. Many popular music APIs provide JSON formatted metadata. Some of these APIs include Amazon, Apple Music, Spotify, SoundCloud, and others.

The JSON format eases development because it is object-oriented and easier than XML to parse since JSON documents are comprised of a comma-separated list of one or more key-value pairs. The simplest form of a JSON document is `{key:value}`. You will note that this is the same form as a Python dictionary. From a software development perspective, JSON is well suited to object-oriented programming languages like Python, JavaScript, and others.

Let's consider that we have a simple document case containing an artist name and album name. The document instance would be defined as follows:

```
{"Artist" : "Quincy Jones", "Album" : "Q's Juke Joint"}
```

A group of related documents is referred to as a collection. As an analogy between a relational database and a document database, a table in a relational database is equivalent to a collection in a document database. Each row in a table is equivalent to a document in the collection and each field name (column) in a table is equivalent to a key in a document.

API methods for database migration

Now that we know what a document store is, I'll show how to use a Python API to migrate a relational database to a document database. My source database was created with MySQL so I used the X DevAPI interface in the MySQL Shell 8.0 application which allows me to access both tables and documents in a database.

To accomplish my goal, I'll use table methods to access data in tables, and document methods to build and verify my documents. Three steps are required to migrate data: 1) create a collection, 2) fetch rows from a table, and 3) add fetched rows to the document collection.

Table methods:

```
table.select() # Returns a data set with rows

result.fetch_one_object() # Returns a dictionary/json object
```

Document methods:

```
collection.add() # Inserts a document into a collection
collection.find() # Returns data set with all documents in a collection
```

The process

The three steps that I identified above assume that the relational database exists, that a database connection has already been established, the source table exists. Before starting the migration process, I’m assuming that a connection to the database has already been made and that the following instances already exist:

```
my_session = mysqlx.get_session( <URI> ) # Session instance
my_database = my_session.get_schema('my_db')
```

The first step in the migration process is to create the target collection.

Step 1: Create collection

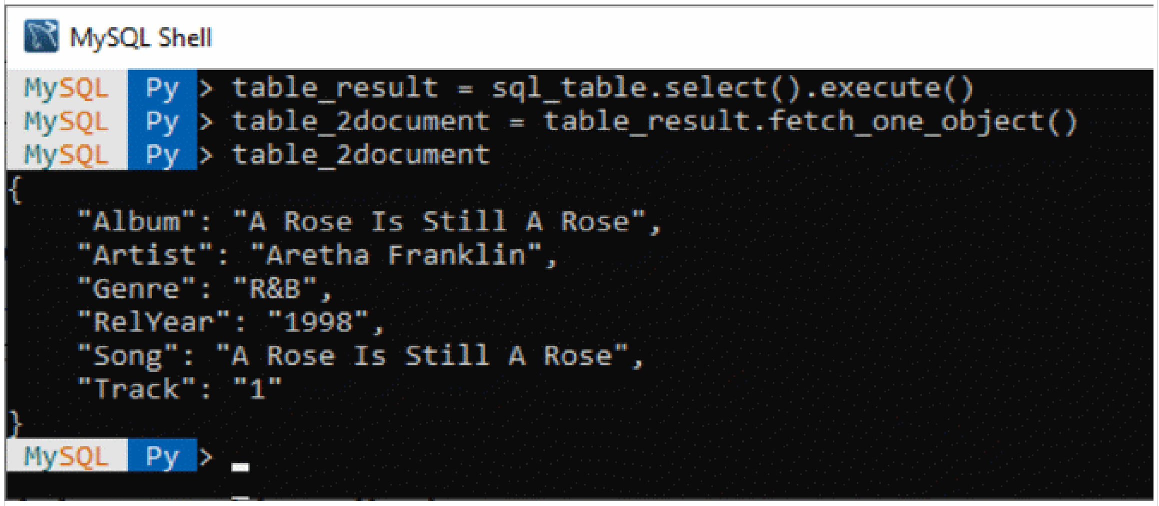
```
doc_collection1 = my_database.create_collection('my_collection1')
```

In the statement above, I’ve created a collection object named `my_collection1` using the `create_collection()` method and assigned it to `doc_collection1` which will be the target document collection. I will subsequently add, update, or remove documents as necessary.

Step 2: Fetch table row data

In the next step, I need to extract my metadata from the source SQL database. This metadata is in a table named `sql_table`. To extract data in a table row, I execute the statements below.

```
table_result = sql_table.select().execute()
table_2document = table_result.fetch_one_object()
```



```
MySQL  Py > table_result = sql_table.select().execute()
MySQL  Py > table_2document = table_result.fetch_one_object()
MySQL  Py > table_2document
{
    "Album": "A Rose Is Still A Rose",
    "Artist": "Aretha Franklin",
    "Genre": "R&B",
    "RelYear": "1998",
    "Song": "A Rose Is Still A Rose",
    "Track": "1"
}
MySQL  Py > _
```

Figure 2: Fetch a JSON document from a SQL table

In the code above, the `select()` method is analogous to the SELECT statement in SQL. It returns a result that is a list of rows. Next, I need to fetch each row, convert it to a JSON object, and add it to my collection. The `fetch_one_object()` method fetches a row from the table as a JSON object.

The `table_2document` result object shows the key:value strings of the metadata fetched from one row of the table. (See Figure 2) While there is a `fetch_one()` method that could fetch a table row, the result is not a JSON object and therefore cannot be added directly to a document. Note that SQL statements executed through the X DevAPI must end with the `execute()` function because they are executed only when that function is called. If omitted, the statement will be ignored.

Step 3: Add document to collection

Each `table_2document` fetched from `my_table` is then added to `doc_collection1` with the `add()` method which adds a JSON document to a collection.

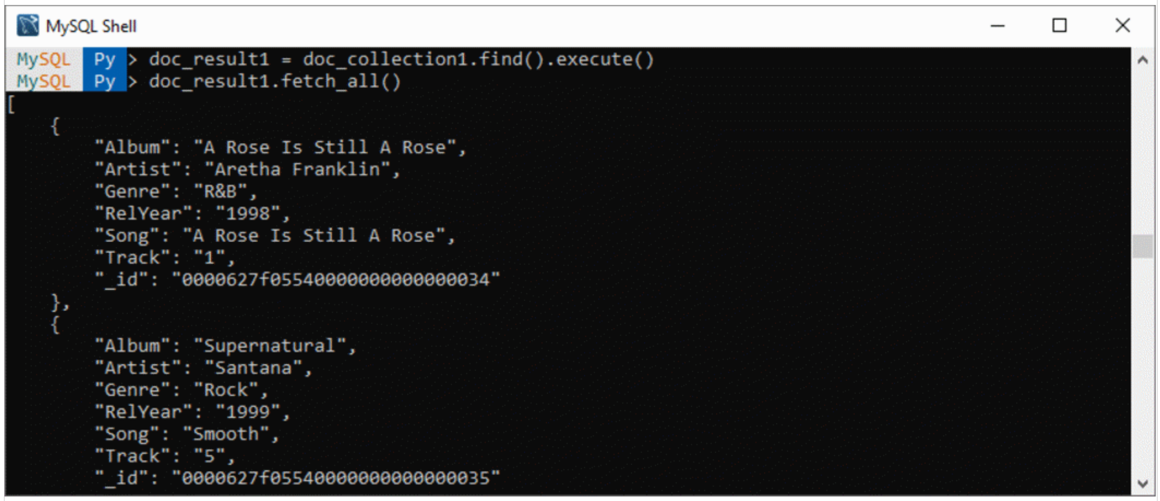
```
doc_collection1.add(table_2document).execute()
```

Build collection

With these three basic steps, I can add a single document to `doc_collection1.` To migrate the entire SQL table to a collection, the code below iterates through each row in the table with the `add()` method.

```
    table_result = sql_table.select().execute()
    table_2document = table_result.fetch_one_object()
    while table_2document:
        doc_collection1.add(table_2document).execute()
        table_2document = table_result.fetch_one_object()
```

The result in Fig. 3 shows two of the five documents added to `doc_collection1`. You will notice that there is an extra `"_id"` field that is automatically added to each document. It is a virtual index that MySQL automatically adds to each document in a collection. See the MySQL 8.0 manual for more information on document indexing.



```
MySQL  Py > doc_result1 = doc_collection1.find().execute()
MySQL  Py > doc_result1.fetch_all()
[
    {
        "Album": "A Rose Is Still A Rose",
        "Artist": "Aretha Franklin",
        "Genre": "R&B",
        "RelYear": "1998",
        "Song": "A Rose Is Still A Rose",
        "Track": "1",
        "_id": "0000627f05540000000000000034"
    },
    {
        "Album": "Supernatural",
        "Artist": "Santana",
        "Genre": "Rock",
        "RelYear": "1999",
        "Song": "Smooth",
        "Track": "5",
        "_id": "0000627f05540000000000000035"
```

Figure 3: Sample of documents added to collection

Document updates

After I have migrated my SQL table to the document collection, I will continue to either build on it with new metadata documents, update incorrect or incomplete documents, or remove documents. As we have already seen, I can use the `add()` method to add new documents to my library or simply add new fields (key-value pairs), effectively changing the schema on-the-fly.

If for example I need to change the spelling of an artist's name, I can use the `doc_collection1.modify()` method. Note that the percent (`%`) wildcard can be used in the search condition string for these methods as illustrated with the `remove()` method in the example below. In addition, note that I have used explicit strings in the `modify()` and `set()` methods to simplify the examples and to keep the focus on function. It is however, good practice to use parameterized placeholders instead of explicit strings.

```
doc_collection1.modify("Artist = 'Santana'").set("Artist", "Carlos Santana")
doc_collection1.remove("Artist like 'Quincy%'")
```

For a more comprehensive list of available create, remove, update, delete (CRUD) methods, see MySQL 8.0 Reference Manual.

Finally

If you want to use Python to automate the creation and maintenance of your music library and want to be able to access music metadata from a wide variety of sources, most deliver content in JSON format. Since document store databases tend to be based on the JSON format or a JSON-like format, and music metadata is widely available in that format, choosing JSON as a preferred format is a logical choice. If your personal music library is in an existing SQL database and you are considering document store as an option, this article may help you in along your migration path.