CS 360
KAIST, Spring 2017
Project: Database and the Web
Part 1: From JSON to SQL
Due Date:

Overview

We provide you with a fairly large volume of data downloaded (over a decade ago!) from the eBay web site and stored in JSON files. JSON is a model primarily used to store semi-structured data. It has become increasing popular in recent years, and, for now, you can treat it as just another type of data that is very straightforward to read. (If you're interested in learning more about JSON, feel free to watch the JSON lecture videos on OpenEdX.)

You will examine the data and design a good relational schema for it. You will then write a Python program to transform the data from its JSON form into SQLite's load file format, conforming to your relational schema. You will create your schema in a SQLite database, load your transformed data, and test it by running some SQL queries over it.

Task A: Examine the JSON data files

We are providing a JSON-encoded auction data set for you to use in your project. The data files are located in the ebay_data.zip

1) As a small data set for initial experimentation and debugging. we suggest you use just one file:
   items-0.json
   It contains 500 auctions, comprising about 900KB of plain-text data.

2) Your system also must work on the full data set, which consists of all 40 files:
   items-n.json for n = 0...39.
   There are a total of about 20,000 auctions, comprising about 38MB of plain-text data.

Your first task is to examine the schema and the JSON files to completely understand the data you will be starting with. You will translate this data into relations. Please read the auction data JSON schema file in items_schema.txt to familiarize yourself with the schema of the data set.

Note that the ItemID attribute is unique and involved in only one auction, while the Name attribute is not unique. One of the most important things to understand about the data you're starting with is that it represents a single point in time. (Specifically, it represents the following point in time: December 20th, 2001, 00:00:01.) It contains items that have been auctioned off in the past and items that are currently up for auction.

Task B: Design your relational schema

Consider the mega-relation that would be formed by simply adding all of the auction data to a single relation, where every attribute in the relation corresponded to a field in the JSON files. Clearly, this would not be a good schema design. You will need to come up with a better schema design.

Create a file design.pdf and document the following:

• Provide your relational schema definitions in text form, including the attributes for each relation. Make sure to clearly indicate your chosen keys (including primary and foreign).

• An Entity-Relation (ER) diagram that describes your schema. Our advice is to use the ER diagram as a starting point to help determine your schema, rather than the other way around. Make sure to include the actual

ER diagram in your design.pdf file.

While designing your relational schema, you may realize that two fields from the JSON files are technically not necessary to have in your schema for it to represent the same information: Currently and Number of Bids. For example, the Number of Bids can be obtained by simply running a query to calculate the number of bids on a particular item. However, for the purposes of this project, we ask that you have both of these attributes somewhere in your schema. For websites with large databases and many users, running a query to calculate an aggregation every time it needs to be viewed can be costly, so storing the aggregated result itself can help improve performance.

Task C: Write a data transformation program

First, make sure you are familiar with the Bulk-Loading Data into SQLite Databases support document, which explains, in great detail, how to appropriately bulk-load data into your database.

Your task is to write a program that transforms the JSON data into SQLite load files that are consistent with the relational schema you designed in Task B. To start you off, we are providing a "skeleton" implementation of the parser in Python – you need to implement the parseJSON method in the starter code, which must parse and extract each JSON file and then output the appropriate SQLite bulk-loading files according to the relational schema you designed in Task B. You are free to add helper functions or modify the files as you see fit. You can find more detailed information in the comments of the starter code:

skeleton_parser.py

You should start by copying this file to a local directory using the cp command; we also encourage you to rename the file to something other than skeleton parser.py to differentiate between the starter code and your own implementation.

The following command will call the skeleton parser on the small data set:

```
python skeleton_parser.py items-0.json
```

To parse the full data set, we simply use the commmand above, but change items-0.json to items-*.json:

```
python skeleton_parser.py items-*.json
```

We strongly suggest that you fully debug your program on the small data set before using it with the full data set.

Please note that some strings in the auction data contain characters commonly used as delimiters. We've cleaned the data to remove all instances of the 'j' character, so feel free to use it as indicated in the bulk-loading support document. Also, open and close files outside any loops that you may have in your parser. This helps to drastically reduce execution time.
For grading purposes, please use the .dat extension for the SQLite load files you generate.

Dollar and date/time values

Dollar value amounts in the auction data are expressed as $X,XXX.XX. To simplify comparisons between dollar values in your database, we have provided a function transformDollar(string) in skeleton_parser.py, in order to reformat these strings into values that SQLite can interpret as floating point numbers. Similarly, date/time values in the auction data are expressed like Mar-25-01 10:25:57. SQLite has support for date/time strings, but only when expressed in the ISO format, e.g. 2001-03-25 10:25:57. We have provided a function transformDttm(string) in skeleton_parser.py in order to do this conversion for you.

Duplicate elimination

When transforming the JSON data to relational tuples, you may discover that you generate certain tuples multiple times but only want to retain one copy. You can code duplicate-elimination as part of your Python parser. Alternatively, you may also use various Unix command-line programs (e.g., sort and uniq) directly on

the generated load files to eliminate the duplicate tuples.

Running time

On the full data set, your parser should take at most a couple of minutes to run. If it takes much longer than that, you are probably doing something unnatural and highly inefficient – please try to reduce the running time before submitting your parser; see the course staff if you need assistance.

Automating the process

Create a file called runParser.sh that consists of the command that invokes your parser over the full data set (i.e. python my_parser.py items-*.json. If there are any other steps involved in your data translation process (such as invoking any Unix commands), also include those steps in the runParser.sh file.

Your runParser.sh must operate on the full data set directly from the class directory, not over your own local copy. When we grade your work up to this point, we will only run sh runParser.sh.

Task D: Load your data into SQLite

SQLite provides a facility for reading a set of commands from a file. You should use this facility for (re)building your database and running sample queries, and you must use it extensively in your submitted work.

Create a command file called create.sql that includes the SQL commands that create all of the necessary tables according to your schema design from Task B. Before creating the tables, you should also ensure that any old tables of the same name have been deleted. (This makes it easier to test your submission.) This file will look something like:

```
drop table if exists Item;
drop table if exists AuctionUser;
...
create table Item ( .... );
create table AuctionUser ( ... );
...
```

Then, create a command file called load.txt that loads your data into your tables. This file will look something like:

```
.separator |
.import items.dat Items
update Items set ... -- Replace all token `NULL' values with null
.import auctionuser.dat AuctionUser
...
```

If you are unsure what these commands do, refer to the Bulk-Loading support document. Each one of your files should run properly when taken as input into the sqlite command, for example:

```
sqlite3 <db_name> < create.sql
```

Task E: Test your SQLite database

The final step is to take your newly loaded database for a test drive by running a few SQL queries over it. As with database creation, first test your queries interactively using the SQLite command-line client, then set up a command file to run them in batch mode. First, try some simple queries over one relation, then more complex queries involving joins and aggregation. Make sure the results look correct. When you are confident that everything is correct, write SQL queries for the following specific tasks:

1. Find the number of users in the database.
2. Find the number of users from New York (i.e., users whose location is the string “New York”).

3. Find the number of auctions belonging to exactly four categories.
4. Find the ID(s) of auction(s) with the highest current price.
5. Find the number of sellers whose rating is higher than 1000.
6. Find the number of users who are both sellers and bidders.
7. Find the number of categories that include at least one item with a bid of more than $100.

Your answers to the above seven queries over the large data set should be (in order): 13422, 80, 8365, 1046871451, 3130, 6717, 150. (Hint: If your second query result is incorrect, it might be because you set the location of a seller incorrectly in your database. The location of a seller is the location of his or her item. If your third query result is wrong, it is probably because you did not eliminate duplicates correctly in your database.)

Your queries should be fairly efficient – they should each take at most ten seconds to execute, most of them under a second. If any of your queries are taking much longer than that, you've probably written them in an unnatural way; please try rewriting them.

Put each of the seven queries in its own command file: query1.sql, query2.sql, ..., query7.sql. Make sure that the naming of your files corresponds to the ordering above, as we will be checking correctness using an automated script.

Submission instructions

To submit Part 1 of the project, first gather the following files in a single submission directory and build a single zip file:

design.pdf
{your_parser_with ID}.py ex)20160000.py
runParser.sh
create.sql
load.txt
query1.sql
query2.sql
query3.sql
quer4.sql
query5.sql
query6.sql
query7.sql

Do NOT include any .dat or .json files in your submission! We reserve the right to deduct points from your project grade if you include them. Once your submission directory is properly assembled, with no extraneous files.

Be sure to submit your zip file "part1.zip" to KLMS for which assignment you're submitting!
You may resubmit as many times as you like; however, only the latest submission and timestamp will be saved, and we will use your latest submission for grading your work and determining any late penalties that may apply. Submissions via email will not be accepted!