Homework B: Viability Testing for TinyDB

Drew Guarnera & Heather Michaud

A development team requires a database for a small store. The database could have up to 1,000 customers with each customer having a purchase history of approximately 500 items. For this task, the development team suggests the python package TinyDB be used as the document based database engine. The mission of the viability test is to assess the potential of using TinyDB for the purposes of a small store customer database.

Requirements

In order to determine if TinyDB is viable for this purpose, we first define a set of database actions which we wish to perform for the small store:

- Add a customer
- Add an item purchased to a customer's 'tab'
- Add multiple instances of the same item purchased to the same customer
- Delete a customer
- Delete an item from a customer
- View a customer and the items they have purchased
- Determine which items are bought most frequently
- Determine which items have been bought least frequently
- Determine the type preference of a customer based on previous item choices
- The database must be capable of containing 500,000 records

The addition, deletion, and viewing of the data for a particular customer or item is crucial for any basic database operations. We expand further upon these requirements by specifying that each of the following requirements must also be met, provided a maximal database of 1,000 customers, each with 500 items:

- Add customer/item takes less than 0.05 seconds
- Deleting a customer/item takes less than 0.1 seconds
- Accessing a customer/item takes less than 0.01 seconds

We expect the most frequent operation that is performed is the accessing of data

within the database for frequently item set mining and other operations to improve the customer experience, so fast access to the data is critical. As we only expect to have 1,000 customers, additions to the database will be less frequent though still relevant. Finally, the deletion of customers or items is expected to happen least frequently so the performance for that is least important.

We also want the API to be easy to use and clean. As this is subjective, we define 'easy to use' such that the API has documentation and examples of each operation we wish to perform. We define 'clean' such that basic operations (add, view, delete) can be performed in one line of code.

Test plan & methodology

We use viability testing to assess whether TinyDB is suitable for the task at hand. We narrow down this list to a set of 10 functional and non-functional requirements. The most commonly used functionality includes adding, deleting, and viewing an item from the database. The most important non-functional requirements are those of a clean design and computationally efficient performance. Thus our specific tests involve verifying the following:

1. An element can be added to the database
2. An element can be added to an item within the database
3. An element can be removed from any location within the database
4. An element can be retrieved from the database
5. A TinyDB can consist of 1,000 elements which have 500 sub-elements
6. Adding element takes less than 0.05 seconds
7. Deleting an element takes less than .1 seconds
8. Accessing an element takes less than 0.01 seconds
9. Documentation exists and is easily accessible and thorough
10. Add, view, and delete can be performed in less than one LOC

To accomplish these testing tasks, python and TinyDB are used primarily. Timing performance is compared to that of SQLite. For the last two requirements, manual verification is used.

Prototype

```
import time
from tinydb import TinyDB, where
import json
import unittest

db = TinyDB("data.json")

class TinyDBAdd(unittest.TestCase):

    def setUp(self):
        db.insert( { 'customerid': 1,
         'items': [ {'name': 'raspberry pi', 'price': 30},
          {'name': 'pi case', 'price': 8} ] } )
        db.insert( { 'customerid': 2,
         'items': [ {'name': 'touch screen', 'price': 90},
         {'name': 'bluetooth', 'price': 15} ] } )

    def tearDown(self):
        db.purge()
        db.close()

    def test_tinydb_add_item(self):
        # Check that new item doesn't exist for customer 1
        self.assertEqual(db.count((where('customerid') == 1) &
        (where('items').any(where('name') == 'battery'))), 0)

        # Grab customer 1 data
        result = db.search(where('customerid') == 1)

        # Grab customer 1 element info
        element = db.get(where('customerid') == 1)

        # Insert a new item and save it
        result[0]['items'].append( {'name': 'battery', 'price': 1} )
        db.update(result[0], eids=[element.eid])

        # Check that new item now exists
        self.assertEqual(db.count((where('customerid') == 1) &
        (where('items').any(where('name') == 'battery'))), 1)

if __name__ == '__main__':
    unittest.main(warnings='ignore')
```