COM661 Full-Stack Strategies and Development

Practical C4: Dataset Preparation

Aims

- To identify potential formats for source datasets
- To present potential sources for acquisition of suitable datasets
- To introduce the Yelp Open Dataset as the basis for a demonstration API
- To demonstrate techniques for dataset preparation and manipulation
- To identify and eliminate inconsistencies in datasets
- To integrate a new dataset into the previously implemented API

Contents

C4.1 DATASET SELECTION 2

C4.1.1 Formats and Challenges 2
C4.1.2 Desirable Characteristics 2
C4.1.3 Sample Datasets – Starting points for research 3
C4.1.4 Introducing the Yelp Open Dataset 4

C4.2 DATASET MANIPULATION 6

C4.2.1 Dataset Design 6
C4.2.2 Create and Merge Collections 8
C4.2.3 Handling Inconsistencies 12

C4.3 DATASET INTEGRATION 20

C4.3.1 Top-level Documents 21
C4.3.2 Sub-documents 24

C4.4 FURTHER INFORMATION 25

C4.1 Dataset Selection

So far, we have constructed our API around a randomly generated dataset of businesses and reviews, but it is important to be able to deploy the techniques illustrated so far for any data collection that we may wish to make available. In this Practical, we will identify some of the desirable characteristics for datasets, introduce some resources from which suitable datasets may be retrieved, and provide a worked example of the deployment of a real-world dataset to the API that we have developed so far.

Note:	This practical is essentially preparation for the assignment in this module. The steps illustrated here are those that you will need to follow for your own datasets – but you are strongly encouraged to also complete the steps for the dataset introduced here as (i) practice in the activity, and (ii) to provide a back-end platform to support the examples to follow in Section D of the module material.

C4.1.1 Formats and Challenges

Datasets are available in many forms. They may be presented as an Excel (or other) spreadsheet, as comma-separated values (.csv), as XML, as plain text files, or in some other format. However, as we are implementing our back-end database in MongoDB, by far the easiest format for us to work with is JSON. As well as corresponding exactly to the MongoDB structure, JSON also correlates with the Python dictionary structure, ensuring that data to be sent to or retrieved from the database can be easily manipulated by our program code.

If a desirable dataset is only available in a format other than JSON, then there are a wide variety of online tools that will easily handle the conversion. Some of these are linked in Section C4.4, but a Google search of (e.g.) “XML to JSON conversion” will return a range of alternatives.

C4.1.2 Desirable Characteristics

To demonstrate a flexible and functional API there are a few desirable features of any dataset that you may select.

i) It should contain sufficient top-level documents to enable the demonstration of a full range of functionality. For example, if the dataset is based around campuses of Ulster University, then there are only 4 top-level documents to maintain. These can be easily displayed on a single page, so limiting options for demonstration of pagination and searching.

ii) The dataset should contain at least one second-level element, such as the **`reviews`** sub-document collection in the **`biz`** example. The manipulation of sub-documents will enable you to demonstrate a much greater range of functionality in your applications.

iii) Since one of the principles of MongoDB is the duplication of data where it aids performance of the most frequently performed operations, then the provision of multiple collections can help to demonstrate this. For example, in the **`biz`** example, we record the username of the person that has contributed a review. A separate **`users`** collection may then contain more information on that user.

C4.1.3 Sample Datasets – Starting points for research

There are many freely available datasets on the Internet that would make excellent vehicles for your own applications, but you may also choose to construct your own dataset from some of the randomisation techniques already presented. The following list presents potential starting points for sourcing a dataset for use in your assignment application.

Awesome JSON Datasets
https://github.com/jdorfman/awesome-json-datasets
Wide range of datasets on various topics including climate, crime, TV shows, movies and more

Data.gov
https://catalog.data.gov/dataset?res_format=JSON
16315 Datasets from US Government Departments

Data.gov.ie
https://data.gov.ie/dataset?res_format=JSON
Houses of the Oireachtas Open Data APIs

FreeCodeCamp.org
https://www.freecodecamp.org/news/https-medium-freecodecamp-org-best-free-open- data-sources-anyone-can-use-a65b514b0f2d/
Datasets from a wide range of providers including World Health Organisation, World Bank, EU, U.S., Census and others

OpenDataNI
https://www.opendatani.gov.uk/dataset?res_format=JSON
Local open-source data sets

Forbes.com
https://www.forbes.com/sites/bernardmarr/2016/02/12/big-data-35-brilliant-and-free- data-sources-for-2016/#20107990b54d
Directory of data providers

Datahub.io
https://datahub.io/collections
Collection of datasets on topics including football, climate change, education, TV and movies, economics and many more

Data.europa.eu
https://data.europa.eu/euodp/en/home
European Union Open Data

Open.canada.ca
https://open.canada.ca/en
Government of Canada Open Data Portal

Kaggle
https://www.kaggle.com/datasets
Wide range of datasets to download

OpenWines
http://openwines.eu/en/datasets.html
Vineyards, grapes and wines

UK Parliament Data
http://www.data.parliament.uk/dataset
Political and parliamentary data

OpenFootball Data
https://github.com/openfootball/football.json
Data from the 2010/11 season to the present day

Oscar winners and nominations
https://datahub.io/rufuspollock/oscars-nominees-and-winners
Data from 1927 to present day

Consumer Data Research Centre
https://data.cdrc.ac.uk
Collections of civic data from the UK

Transport for London
https://tfl.gov.uk/info-for/open-data-users/our-open-data
Collection of TfL Data Feeds

Translink
https://www.opendatani.gov.uk/organization/translink
Northern Ireland Public Transport Data

Northern Ireland Assembly
http://data.niassembly.gov.uk
Open Data Collections from the NI Assembly

Million Song Dataset
http://millionsongdataset.com/lastfm/
Provided by last.fm

100 Best Books
https://github.com/benoitvallon/100-best-books/blob/master/books.json
JSON data for the 100 best classical books of all time

Scottish Parliament Data
https://data.parliament.scot/#/datasets
121 datasets available

Socrata Open Data
https://dev.socrata.com
A range of open data resources from governments, non-profits, and NGOs around the world

C4.1.4 Introducing the Yelp Open Dataset

Yelp is an internet company that publishes crowd-sourced reviews about businesses. It was founded in 2004 and has since grown into a community of almost 100 million unique users per year across its website and mobile platforms and a collection of over 200 million reviews on businesses. Yelp makes a large subset of this data available as an open-source resource for personal, academic and research purposes, covering over 8.5 million reviews by over 2 million users on 160,000 businesses across 6 collections as follows.

`busineses`	A collection of details on businesses, including location data, attributes and categories.
`reviews`	Review information including a free text review, a rating in the range 1-5 stars and links to the business the review is written for and the user that contributed the review
`users`	Data associated with the user including a list of that user's "friends".
`checkins`	A list of timestamps of checkin activity for each business.
`tips`	Tips contributed by a user for a business. Tips tend to be shorter than reviews and convey quick suggestions or comments
`photos`	A collection of photos, each described by a caption, a category and a link to the business to which it relates.

The collection is made available as 6 JSON files, one for each of the collections described above, as well as a set of images. It can be downloaded free from https://yelp.co.uk/dataset.

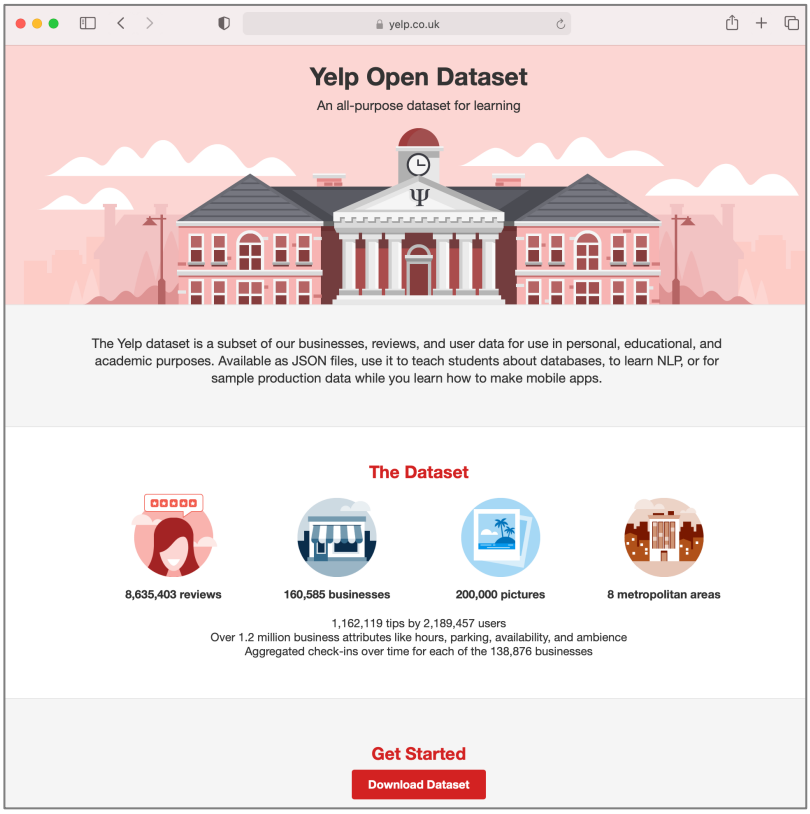

Figure C4.1 The Yelp Open Dataset

In this section, we will use the Yelp Open Dataset as a vehicle to explore the issues in obtaining a dataset, manipulating it into a form suitable for our chosen application and deploying it as the core to an API to enable retrieval, manipulation and searching.

Note:	You should feel free to download the dataset for yourself and experiment with the manipulation techniques discussed in this section. However, the final version of the dataset is also provided for you in the files ***businesses.json*** and ***users.json***. These can be imported to the database by the following commands
	
	`mongoimport --db yelpDB --collection businesses --jsonArray businesses.json`
	
	`mongoimport --db yelpDB --collection users --jsonArray users.json`

Do it now!	Examine the documentation for the Yelp Open Dataset at https://www.yelp.co.uk/dataset/documentation. Examine the fields provided in each collection, remembering that not all fields will necessarily be included in each document. Consider any modifications that you might need to make to the structure to have it better suited to a MongoDB database – we will identify and complete some of these modifications in the next section.

C4.2 Dataset Manipulation

We have seen many resources from which suitable datasets can be obtained, but data is presented in a wide variety of forms and it is often required to perform some manipulation to best prepare it for your application. This section deals specifically with the Yelp Open Dataset, but many of these techniques will be equally applicable to the data that you will source for your own applications.

C4.2.1 Dataset Design

The initial step is to compare the structure of the data provided to the structure required for your application. We have already seen how a characteristic of MongoDB databases is the nested structure that allows all the information that is required to be retrieved by a single query. When displaying details of a business, therefore, it is reasonable to assume that we might wish to display the general business information (name, address, category, etc.) along with (perhaps the most recent) reviews, tips and checkins, and possibly illustrating the page with one or more relevant photos. To accommodate this, we want to modify the structure of the businesses collection to add sub-collections for reviews, tips, checkins and photos relevant to that business. This will allow us to drop the explicit collections for reviews, tips, checkins and photos, leaving only (the expanded) businesses and users collections in our application.

In addition, there are other modifications that we might wish to make. If we choose not to make use of the entire collection of businesses (for example to reduce the preparation time required or to make the collection more portable), we can drop any photos related to businesses that we are not going to include. In addition, if we select a subset of businesses, we may now have users who have only contributed reviews or tips to businesses that do not feature in our collection, so they can also be dropped. Finally, we see that the businesses and users collections already contain unique **`business_id`** and **`user_id`** fields. However, these are not MongoDB **`_id`** values, so we will want to replace these with the "real" **`_id`** fields when referring to businesses and users from documents in our collections.

We will construct a Python application to automate the transformation of the dataset in two phases as follows:

i) First, we will create the database, import all of the collections of the Yelp Open Dataset and integrate the reviews, tips, checkins and photos into the businesess collection.

ii) Then, we will perform any required modifications to the combined data to best prepare it for use in our API.

As an initial step, we prepare the skeleton of our application ***make_dataset.py*** which consists of calls to the various functions that we will implement. The following sections will discuss each of these functions in turn.

Note:	If you plan to follow this implementation for yourself, be aware that some of the following steps will require a very long execution time (measured in hours and sometimes even more than 1 day!). The best approach is to comment out all of the function calls in the code below, adding the functions one at a time and uncommenting (only) the newly added function so that a different function is run each time you execute the application. For example, on the first execution, only run the **`create_database()`** function; on the second execution, only run the **`add_reviews_to_business()`** function and so on.

File: biz/make_dataset.py

```
from pymongo import MongoClient
import json
from pathlib import Path
from bson import ObjectId

# functions will go here...

create_database()
add_reviews_to_businesses()
add_photographs_to_businesses()
add_checkins_to_businesses()
add_tips_to_businesses()
drop_unwanted_collections()

fetch_required_photos()
purge_inactive_users()
make_locations_GeoJSON()
fix_user_ids_in_businesses()
fix_user_ids_in_users()
```

C4.2.2 Create and Merge Collections

The first step is to create a collection for each of the JSON files in the Yelp Open Dataset. This could be achieved by use of the **`mongoinsert`** command if the data items were presented as a JSON array, but in this case the data is provided as one document per line in the file, so the easiest method is to read the file in one line at a time, convert the line of text to a JSON object and then issue the **`insert_one()`** command to add that document to the relevant collection. Here we see the code to create the **`businesses`** and **`users`** collections, but exactly the same technique is also used for **`reviews`**, **`photos`**, **`checkins`** and **`tips`**.

File: biz/make_dataset.py

```
def create_database():
    client = MongoClient("mongodb://127.0.0.1:27017")
    db = client.yelpDB

    businesses = db.businesses
    with open('yelp_academic_dataset_business.json') as f:
        for line in f:
            businesses.insert_one(json.loads(line))
    print("Businesses loaded")

    users = db.users
    with open('yelp_academic_dataset_users.json') as f:
        for line in f:
            users.insert_one(json.loads(line))
    print("Users loaded")

    # repeated for 'reviews', 'photos', 'checkins' and 'tips'
```

At this stage, we have a database consisting of 6 collections, corresponding to the 6 JSON files supplied in the Yelp Open Dataset. To integrate the collections into a usable MongoDB structure, we first merge the reviews into the businesses by traversing the businesses collection in the following manner.

1. For each business in the collection
    a. Initialise a **`reviews`** collection as an empty list
    b. For each document in the **`reviews`** collection matching the current business
        i. Add that review to the list
    c. Update the current business to add the **`reviews`** field as a list of sub-documents as well as a **`review_count`** field as the number of reviews in the list

This process is described by the following code.

File: biz/make_dataset.py

```
def add_reviews_to_businesses():
    client = MongoClient("mongodb://127.0.0.1:27017")
    db = client.yelpDB
    businesses = db.businesses
    reviews = db.reviews

    num_processed = 0
    for business in businesses.find(
           {}, { "business_id":1 }, no_cursor_timeout = True):
        reviews_of_business = []
        for review in reviews.find(
                { "business_id" : business["business_id"] } ):
            reviews_of_business.append(review)

        businesses.update_one(
                { "business_id" : business["business_id"] } ,
                { "$set" : { "reviews" : reviews_of_business,
                "review_count" : len(reviews_of_business) } } )
        num_processed = num_processed + 1
        if num_processed % 1000 == 0:
            print(str(num_processed) + " businesses processed")
```

Note that the code also maintains a count of the number of businesses processed so far and reports the progress after every 1000 businesses. This allows us to monitor progress and estimate the time that will be required to process every business. We will include similar code in the remaining functions – though it will not be included in the code boxes presented.

Note: If the execution time appears to be excessive, you can abort this function once sufficient businesses have been processed. The remaining functions will iterate for each business, so will automatically adjust to cater only for those businesses processed at each stage.

If this function is aborted, you should delete un-processed businesses by removing those that do not contain a **`reviews`** field. This can be achieved from the MongoDB shell prompt by the command **`db.businesses.remove( {"reviews": {"$exists":false} } )`**

Having merged the reviews into the businesses, we now want to perform a similar merge for the photos. Each document in the photos collection includes a **`business_id`** field that

connects the photo to an entry in the businesses collection. Remember that although the **business_id** is not a valid MongoDB **_id** value, it can still be used as the key field in a search. The process is similar to that for reviews except that for each photo found matching a particular **business_id**, we create a **photo_obj** object with fields **filename**, **caption** and **label** that we initialise to the values retrieved from the **photos** collection. This is essentially the same as simply adding each **photo** document returned by the query but illustrates how we can be selective about the fields that we merge, simply omitting any that we do not want.

File: biz/make_dataset.py

```
def add_photos_to_businesses():
    client = MongoClient("mongodb://127.0.0.1:27017")
    db = client.yelpDB
    businesses = db.businesses
    photos = db.photos

    for business in businesses.find(
           {}, { "business_id":1 }, no_cursor_timeout = True):
        photos_of_business = []
        for photo in photos.find(
                { "business_id" : business["business_id"] } ):
            photo_obj = { "filename" :
                                  photo["photo_id"] + ".jpg",
                          "caption" : photo["caption"],
                          "label" : photo["label"]
                        }
            photos_of_business.append(photo_obj)

        businesses.update_one(
              { "business_id" : business["business_id"] } ,
              { "$set" : { "photos" : photos_of_business,
                   "num_photos" : len(photos_of_business) } } )
```

Next, we can merge the information from the **checkins** collection into **businesses**. This collection comprises one document per business, with the **business_id** value and a string containing a list of timestamps, separated by commas. In this case, the timestamps are also prefaced by a space character, so to make the timestamps more easily usable in a search function, we process each timestamp list by separating into a list of individual string values which we then assign to the **checkins** field for the business.

Note also the **`no_cursor_timeout = True`** parameter added to the **`businesses.find()`** query. Where we have nested queries, the activity in the inner loop may cause the outer loop to time out – for example if the time taken to process the checkins of a business exceeds the timeout period of the **`businesses.find()`** query. In such cases we can adjust the timeout period or (as we do here) remove it completely.

File: biz/make_dataset.py

```
def add_checkins_to_businesses():
    client = MongoClient("mongodb://127.0.0.1:27017")
    db = client.yelpDB
    businesses = db.businesses
    checkins = db.checkins

    for business in businesses.find(
           {}, { "business_id":1 }, no_cursor_timeout = True):
        checkins_of_business = []
        for checkin in checkins.find(
                { "business_id" : business["business_id"] } ):
            checkins_of_business = checkin["date"].split(',')
        checkins_of_business = [
                  c_i.strip() for c_i in checkins_of_business ]
        businesses.update_one(
           { "business_id" : business["business_id"] } ,
           { "$set" : { "checkins" : checkins_of_business,
                        "checkin_count" :
                                len(checkins_of_business) } } )
```

The final stage in our merge operation is to integrate the information in the **`tips`** collection with **`businesses`**. Again, we adopt the same process – visiting each of the businesses in turn, constructing a list of tips for that business by retrieving from the **`tips`** collection those documents that match the current **`business_id`**, and updating the business to add both the **`tips`** field and a **`tip_count`** field.

File: biz/make_dataset.py

```
def add_tips_to_businesses():
    client = MongoClient("mongodb://127.0.0.1:27017")
    db = client.yelpDB
    businesses = db.businesses
    tips = db.tips

    for business in businesses.find(
            {}, { "business_id":1 }, no_cursor_timeout = True):
        tips_of_business = []
        for tip in tips.find(
                { "business_id" : business["business_id"] } ):
            tips_of_business.append(tip)

        businesses.update_one(
            { "business_id" : business["business_id"] } ,
            { "$set" : { "tips" : tips_of_business,
                         "tip_count" : len(tips_of_business) } } )
```

Now that the required content from **`reviews`**, **`photos`**, **`checkins`** and **`tips`** has been merged into the **`businesses`** collection, we can remove the collections that we no longer need by dropping them from the database. This could easily be done from the MongoDB Shell prompt, but it is just as convenient to add a function to our application.

File: biz/make_dataset.py

```
def drop_unwanted_collections():
    client = MongoClient("mongodb://127.0.0.1:27017")
    db = client.yelpDB

    db.reviews.drop()
    db.photos.drop()
    db.checkins.drop()
    db.tips.drop()
```

C4.2.3 Handling Inconsistencies

At this stage, our database consists of 2 collections – **`businesses`** and **`users`**, where each business entry contains sub-collections of **`reviews`**, **`photos`**, **`checkins`** and **`tips`** as well as the business name, location, category and other information. However, there exists the potential for inconsistencies and optimisations in the data – especially if we aborted the initial stage before all businesses were processed. In this section we will identify and remedy some of these issues.

As well as the JSON data, the Yelp Open Database contains a collection of images corresponding to the contents of the **`photos`** collection. If we have chosen to use only a subset of the businesses, we will now be left with some photos relating to businesses that are not present in our database, so it is worth processing the **`photos`** collection so that only those relating to "real" businesses are kept. The following code for **`fetch_required_photos()`** visits each photo related to a businesses in the collection and moves the corresponding image files to a new **./photos/** folder. From there, we can host the photos that have been moved to the new location on a web server so that they can be easily linked from the front-end application.

File: biz/make_dataset.py

```
def fetch_required_photos():
    client = MongoClient("mongodb://127.0.0.1:27017")
    db = client.yelpDB
    businesses = db.businesses
    users = db.users

    for business in businesses.find(
                {}, { "photos" : 1 }, no_cursor_timeout = True):
        for photo in business["photos"]:
            Path("../yelp_photos/photos/" +
                  photo["filename"]).rename(
                                "./photos/" + photo["filename"])
```

Note:	The photos required in the version of the completed database that is supplied for you are hosted on a web server at http://adrianmoore.net/yelp_photos/. You can access them directly from here in your application rather than hosting them yourself.

Try it now!	Write the Python program ***get_business_images.py*** that selects a business at random from the **`businesses`** collection and prints to the console the business name and the full URL (on the adrianmoore.net server) of each photo belonging to that business. Copy the URLs and paste into the address bar of a web browser to verify that the URLs are being retrieved and constructed correctly.

To this point, we have not modified the **`users`** collection in any way, but we need to recognise that if we have chosen to use only a subset of the businesses in the original collection, we may now have users who have not contributed reviews or tips for any of the businesses that remain. For consistency, these users should now be identified and removed.

One way to achieve this is to construct sets of active and inactive users, where an active user is defined as one for whom the **user_id** field appears somewhere in the **businesses** collection – either as a contributor of a review or of a tip. This is illustrated by steps 1 and 2 in the code for **purge_inactive_users()** below. First (step 1), we iterate across the collection of businesses, adding to the **active_users** set the **user_id** of any contributor of a review or of a tip for that business. Then, (step 2) we iterate across the documents in the **users** collection, adding to the **inactive_users** set the **user_id** of any user that does not appear in the **active_users** set.

File: biz/make_dataset.py

```
def purge_inactive_users():
    client = MongoClient("mongodb://127.0.0.1:27017")
    db = client.yelpDB
    businesses = db.businesses
    users = db.users

    # 1. get the set of active users
    active_users = set()
    for business in businesses.find(
             {}, {"reviews":1, "tips":1},
             no_cursor_timeout = True ):
        for review in business["reviews"]:
            active_users.add(review["user_id"])
        for tip in business["tips"]:
            active_users.add(tip["user_id"])

    # 2. get the set of inactive users
    inactive_users = set()
    for user in users.find( {}, {"user_id":1},
                            no_cursor_timeout = True ):
        if user["user_id"] not in active_users:
            inactive_users.add(user["user_id"])
```

Now that we have separate sets of active and inactive users, we can modify the contents of the **users** collection so that only the users in the active set are included. We can either achieve this by insertion or deletion – for the latter, we would traverse the set of inactive users and remove the entry for each user found, but instead we choose here to drop the entire collection and re-build it from the original JSON file, this time only issuing the **insert_one()** query for users that appear in the **active_users** set.

File: biz/make_dataset.py

```
def purge_inactive_users():

    ...

    # 3. remove inactive users from the collection
    users.drop()
    with open('yelp_academic_dataset_user.json') as f:
        for line in f:
            this_user = json.loads(line)
            if this_user["user_id"] in active_users:
                users.insert_one(this_user)
```

Where we have removed users from the collection, we now need to make sure that those users are not contained in the **`friends`** list for any user that remains. The Yelp Open Dataset documentation describes the **`friends`** list as a list of **`user_id`** values – but that is actually not the case. Instead, the **`friends`** field is a single string of **`user_id`** values, separated by commas and with embedded spaces. Our first task, therefore, is to create a **`friends_of_user`** list by splitting the value of the **`friends`** field on the comma character. Then, we create a replacement **`friends_of_user`** list by traversing the original list and stripping the entry of any whitespace characters. Next, we can iterate across **`friends_of_user`**, checking for each **`user_id`** that it is in the set of **`active_users`** and adding it to an **`active_friends`** list if it is found. Finally, we issue an **`update_one()`** query to the **`users`** collection, updating that user's **`friends`** field to the list of active friends. This process is implemented in Stage 4 of the **`purge_inactive_users()`** function below.

File: biz/make_dataset.py

```
def purge_inactive_users():

    ...

    # 4. remove inactive users from friends lists
    users_list = list( users.find({},
                       {"user_id":1, "friends":1} ) )
    for user in users_list:
        active_friends = []
        friends_of_user = user["friends"].split(',')
        friends_of_user = [
                           f.strip() for f in friends_of_user ]
        for friend in friends_of_user:
            if friend in active_users:
                active_friends.append(friend)
        db.users.update_one(
            { "user_id" : user["user_id"] },
            { "$set" : { "friends": active_friends } } )
```

Try it now!	Write the Python program ***get_friends.py*** that selects a user at random from the **`users`** collection and prints to the console the names of that user's friends. Hint: This will require queries on both the **`businesses`** and **`users`** collections.

We have seen in Practical C3 how the MongoDB Aggregation Pipeline includes support for Geospatial queries where the collection provides location information in GeoJSON form. The **`businesses`** collection in the Yelp Open Dataset does provide location information for each business, but as separate **`longitude`** and **`latitude`** values rather than as a GeoJSON **`Point`**. We can remedy that by implementing the function **`make_locations_GeoJSON()`** which visits each business in the collection and creates a **`geo_point`** object in the required format, with a **`type`** field of **`Point`** and a **`coordinates`** list containing the longitude and latitude values. We can then issue an **`update_one()`** query to the business specifying the new field **`location`** as the **`geo_point`** object and also unsetting (deleting) the original **`longitude`** and **`latitude`** fields.

File: biz/make_dataset.py

```
def make_locations_GeoJSON():
    client = MongoClient("mongodb://127.0.0.1:27017")
    db = client.yelpDB
    businesses = db.businesses

    for business in businesses.find(
           {}, {"business_id":1, "latitude":1, "longitude":1},
           no_cursor_timeout = True ):
        geo_point = { "type" : "Point",
                      "coordinates" : [business["longitude"],
                                       business["latitude"]] }
        businesses.update_one(
             { "business_id":business["business_id"] },
               [ { "$set" : { "location" : geo_point } },
                 { "$unset" : ["longitude", "latitude"]}
               ])
```

In performing the modifications described in this section, we have made use of the **`user_id`** field provided in the original dataset. Since these are not valid MongoDB **`_id`** values and each **`user`** document also contains a **`_id`** field that will have been automatically generated during the initial import into the database, we should now replace all references to **`user_id`** values with their **`_id`** equivalents.

First, we consider the **`user_id`** field referenced in each review or tip that has been left for a business. The technique is to iterate across each business in the collection, fetching the reviews and tips for each. Then, for each review, we retrieve from the **`users`** collection the **`_id`** of the document that contains the **`user_id`** value matching that found in the review. We then replace the **`user_id`** value in the review with the **`_id`** of that user. We then repeat the process for the tips of the business before updating the **`reviews`** and **`tips`** fields to the new values. Note that we can also remove the **`business_id`** fields from both **`reviews`** and **`tips`** as these are now embedded inside the business to which they belong (as well as the redundant **`review_id`** field in the **`review`** object).

File: biz/make_dataset.py

```
def fix_user_ids_in_businesses():
    client = MongoClient("mongodb://127.0.0.1:27017")
    db = client.yelpDB
    businesses = db.businesses
    users = db.users

    businesses_list = list(
              businesses.find( {}, {"reviews":1, "tips":1} ) )
    for business in businesses_list:
        business_id = str(business["_id"])

        for review in business["reviews"]:
            this_user = review["user_id"]
            user = users.find_one(
                              {"user_id":this_user}, {"_id":1})
            review["user_id"] = str(user["_id"])
            del review["review_id"]
            del review["business_id"]

        for tip in business["tips"]:
            tip_id = str(tip["_id"])
            this_user = tip["user_id"]
            user = users.find_one(
                              {"user_id":this_user}, {"_id":1})
            tip["user_id"] = str(user["_id"])
            del tip["business_id"]

        businesses.update_one(
            { "_id" : ObjectId(business_id)},
            { "$set" : { "reviews" : business["reviews"],
                         "tips" : business["tips"]
                       }
            } )
```

We can now repeat this process for references to **`user_id`** within the **`users`** collection. This occurs in the specification of the **`friends`** field, which is a list of **`user_id`** values. Our task is therefore to replace this with the matching list of **`_id`** values.

Since any user might appear in the **`friends`** list of multiple other users, it makes sense to generate a mapping table to convert **`user_id`** values to **`_id`** values rather than convert each **`user_id`** every time it is found. The mapping table is easily implemented as a dictionary which is built by retrieving the **`_id`** and **`user_id`** values for all users and setting

each dictionary entry with a key field of the **user_id** and a value of the corresponding **_id**. Hence the dictionary will have a form such as

```
{
   “user_id 1” : “_id 1”,
   “user_id 2” : “_id 2”,
   ...
}
```

Once the mapping table has been constructed as the dictionary **users_dict**, we iterate across the entries in the **users** collection and traverse the **friends** list for each, generating a **new_friends** list by converting the **user_id** to the corresponding **_id**. We then issue an **update_one()** query for the user, replacing the current value of **friends** with the **new_friends** list of user **_id** values.

File: biz/make_dataset.py

```
def fix_user_ids_in_users():
    client = MongoClient("mongodb://127.0.0.1:27017")
    db = client.yelpDB
    users = db.users

    users_dict = {}
    users_list = list(
                      users.find({}, {"_id":1, "user_id":1} ) )
    for user in users_list:
        users_dict[user["user_id"]] = str(user["_id"])

    users_list = list(
                      users.find({}, {"_id":1, "friends":1} ) )
    for user in users_list:

        this_user = str(user["_id"])
        new_friends = []
        for friend in user["friends"]:
            new_friends.append( users_dict[friend] )

        users.update_one(
             { "_id" : ObjectId(this_user) },
             { "$set" : { "friends" : new_friends } } )
```

Try it now!	At present, each review embedded within a business contains the **`_id`** of the user that left the review, but not their name. Since the reviewer's name would be commonly printed alongside the review, it would be useful to add this field. Write the program ***add_reviewer_names.py*** that visits each review stored in the database and adds the reviewer's name as an additional field called **`"name"`**.

At this point, the **`yelpDB`** database is ready for integration into the API, so the next section in this Practical will revisit the API application from Practical B3 and replace the randomly generated database of business objects with the "real" data.

Note:	Many of the data preparation and transformation steps described in this section could have been implemented more efficiently by carrying out multiple operations at once. For example, when removing the set of inactive users from the friends lists, we could also have swapped the **`user_id`** for **`_id`**. Likewise, the GeoPoint field for a business could have been added while we were merging the reviews. The steps have been described independently here purely in the interests of clarity – when preparing your own datasets you should try to provide as efficient an implementation as possible.

C4.3 Dataset Integration

At this point, we have an API that provides full CRUD facility on a randomly generated database (from Practical C2), as well as a "real-world" database sourced from the Yelp Open Dataset. In this section we will modify the source code of the API to have it operate on the ***yelpDB*** database instead of on our dummy data.

Actually, this is a very straightforward task – the API in its current form already supports all operation on documents and sub-documents, so all that changes with the new database is the field names and the number of sub-document collections provided.

Do it now!	In your ***biz*** project folder, make a copy of ***app.py*** to a file ***old_app.py***. We will now work with ***app.py*** so that it operates on the new database derived from the Yelp Open Dataset.

First, we will modify the API code so that it operates on the ***yelpDB*** database and that the **`businesses`** variable points to the businesses collection contained within it.

Note:	The modification described in this section is made a little easier as the dummy and real datasets both relate to collections of businesses – hence many of the original variable names are still appropriate. You should make sure that all variable names are always appropriate for the subject for which you are providing the application.

File: biz/app.py

```
db = client.yelpDB              # update the database
businesses = db.businesses      # update the collection name
```

C4.3.1 Top-level Documents

Now that we have connected to the updated database and collection, let's examine the code to retrieve the collection of businesses. Here, all that needs to be added is an additional step to repeat the conversion to string of the **_id** field on the **reviews** sub-collection for the sub-collection of **tips**. In addition, we implement exactly the same step for the retrieval of a single business, as shown below.

File: biz/app.py

```
def show_all_businesses():
    ...
    for business in \
        businesses.find().skip(page_start).limit(page_size):
        business['_id'] = str(business['_id'])
        for review in business['reviews']:
            review['_id'] = str(review['_id'])
        for tip in business['tips']:
            tip['_id'] = str(tip['_id'])
        data_to_return.append(business)

    return make_response( jsonify( data_to_return ), 200 )

def show_one_business(id)
    ...
    business = business.find_one( {'_id':ObjectId(id)} )
    if business is not None:
        ...
        for tip in business['tips']:
            tip['_id'] = str(review['_id'])
        return make_response( jsonify(  business ), 200 )
    ...
```

We can show that these modifications have had the desired effect by issuing a GET command to retrieve all business and to retrieve details of a specified business.

Do it now!	Run the modified ***app.py*** application. Now use Postman or a Web browser to make a GET request to http://localhost:5000/api/v1.0/businesses and verify that the endpoint to return all businesses returns a page of information from the new database. Copy the **`_id`** of one of the businesses retrieved and add it to the URL to retrieve only details of that business. Ensure that you receive a response such as that illustrated in Figure C4.2 below.

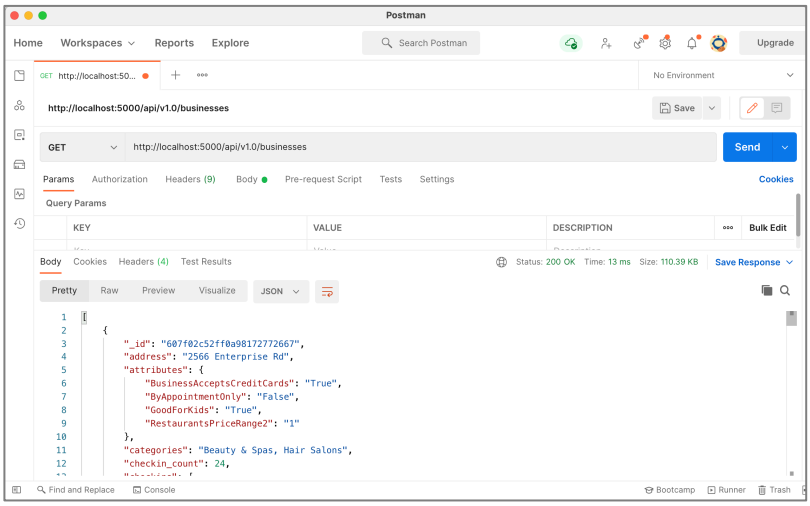

*Figure C4.2 Businesses retrieved from the **`yelpDB`** database*

Next, we consider the modifications that need to be made to the endpoint to add details of a new business. In the original version, the fields that needed to be provided were **`name`**, **`town`** and **`rating`**, while the equivalents in the new database are **`name`**, **`city`** and **`stars`**. In addition, we have added a sub-document collection called **`tips`** for which every document has an **`_id`** field, so we need to initialize **`tips`** to an empty list as is already done for **`reviews`**.

Note that **`reviews`** and **`tips`** are compulsory fields (at present) as the code to retrieve businesses (i.e. **`show_all_businesses()`** and **`show_one_business()`**) explicitly checks for their presence. We could also add empty lists for **`photos`** and **`checkins`**, but as we do not check for them anywhere else in the code, they are not explicitly required.

Note:	Alternatively, we could add code to **`show_all_businesses()`** and **`show_one_business()`** to check that **`reviews`** and **`tips`** exist within the document and only attempt to retrieve them if they are present.

In the API's present form, the modifications required to add a new business are highlighted in the code below.

File: biz/app.py

```
def add_business():
    if "name" in request.form and \
       "city" in request.form and
       "stars" in request.form:
       new_business = {
           "name" : request.form["name"],
           "city" : request.form["city"],
           "stars" : request.form["stars"],
           "reviews" : [], "tips" : []
       }
       ...

def edit_business(id)
    if "name" in request.form and \
       "city" in request.form and \
       "stars" in request.form:
       result = businesses.update_one( \
          { "_id" : ObjectId(id) }, {
            "$set" : { "name" : request.form["name"],
                       "city" : request.form["city"],
                       "stars" : request.form["stars"]
                     }
          } )
       ...
```

Do it now!	Run the modified ***app.py*** application. Now use Postman to make a POST request to http://localhost:5000/api/v1.0/businesses, passing values for the **`name`**, **`city`** and **`stars`** fields. Once you have verified that the new business has been added, make a PUT request to that business to modify some of the parameter values. Finally, make a GET request to the same business to confirm that the values have been changed.

Deleting a business requires no modifications since the only field that is referenced is the **`_id`** identifier. As both the "old" and "new" databases identify a business by its **`_id`**, the existing code will still work as required.

Do it now!	Verify that the API has been updated to operate on the new database by following the following sequence of operations on Postman i) Return a collection of business by issuing a GET request to http://localhost:5000/api/v1.0/businesses ii) Add a new business by a POST request to the same URL, providing field values for **`name`**, **`city`** and **`stars`** iii) Edit the newly added business by issuing a PUT request to the URL returned by the add operation, providing new values for for **`name`**, **`city`** and **`stars`** iv) Delete the new business by issuing a DELETE request to the URL returned by the PUT operation. Issue a GET request to the same URL to ensure that it has been successfully deleted.

C4.3.2 Sub-documents

Fetching the collection of reviews and fetching a single review require no changes since reviews are returned in entirety without reference to any field names. This can be easily verified by using Postman.

Do it now!	Retrieve the details of any single business and add **`/reviews`** to the URL to return the collection of reviews for that business. Now, select one of the reviews and add its **`_id`** field value to the end of the URL to confirm that only that single review is returned.

When adding or editing a review, the "old" version of the API requires the fields **`username`**, **`comment`** and **`stars`**, while the "new" version will require **`user_id`**, **`name`**, **`text`** and **`stars`**. The updates required here are exactly as already seen for adding or editing details of a specified business and are implemented in exactly the same way. The implementation of this is left for you as an exercise.

Note:	The **`name`** field is only required if you have completed the **Do it now!** exercise at the end of Section C4.2 Otherwise, **`name`** can be omitted (and can be retrieved if required by a search on the **`users`** collection) When providing a **`user_id`** value for an add or edit review operation, you can preserve database integrity by selecting one at random from the **`users`** collection.

The final sub-document operation is that which deletes a review from the collection. As for businesses, deleting a review does not refer to any fields other than **`_id`**, which is provided in both the "old" and "new" collections, so the existing code should work perfectly without modification.

Try it now!	Make the modifications required to **`add_new_review()`** and **`edit_review()`**. Test the application using Postman and ensure that all API endpoints are now working as expected with the new database.

C4.4 Further Information

- https://docs.python.org/3/library/json.html
  The Python JSON library

- https://www.freeformatter.com/xml-to-json-converter.html
  XML to JSON Converter

- https://beautifytools.com/excel-to-json-converter.php
  Excel to JSON Converter

- https://www.convertcsv.com/csv-to-json.htm
  CSV to JSON Converter

- https://yelp.co.uk/dataset
  The Yelp Open Dataset

- https://www.yelp.co.uk/dataset/documentation
  Yelp Open Dataset Documentation

- https://orangematter.solarwinds.com/2019/12/22/what-is-mongodbs-id-field-and-how-to-use-it/
  What is MongoDB's **`_id`** Field and How to Use it.

- https://realpython.com/python-pathlib/
  Python 3's **`pathlib`** Module: Taming the File System