Database Workshop

February 25, 2022

Creating a Database

Often it is useful (either so you can construct queries yourself, or so that other people can) to construct your own databases. It is common as small enterprises, departements or projects grow for spreadsheet based data management to become extremely unwieldy and a more robust record keeping methodology becomes needed.

A simple way is to convert the spreadsheet (csv) into a SQL database. We will use the data in the file SE4ALLData.csv as out example. This data is from the world bank https://datacatalog.worldbank.org/dataset/sustainable-energy-all, if you ever need statistics about countries, this is the place to go!

First we connect to the database (creating it if it doesn't already exist). Next we create a table

```
conn = sqlite3.connect("SE4ALL.db")
c = conn.cursor()
c.execute('''CREATE TABLE energy (CountryName TEXT
    NOCASE, IndicatorCode TEXT NOCASE, Year INTEGER,
    Value REAL)''')
conn.commit()
```

The CREATE TABLE keyword is self explanatory. We have declared the column names of the energy table: CountryName, IndicatorCode, Year, Value. Alongside we have declared their type. We now have to read the csv and put the values into the table. First some housekeeping

```
with open("SE4ALLData.csv", 'r') as csvfile:
  reader = csv.reader(csvfile, delimiter=',', quotechar
    ='"')
  headers = next(reader);
  headerid = {};
  for i,h in enumerate(headers):
    headerid[h] = i;

  for row in reader:
      ###update the table
```

The first row of the CSV file contains the header, telling us what is in each column. We use the `next` operator to read it into a list called headers, and then convert the list into a dictionary. Where it says ‘update the table’ we put

```
    for year in range(1990,2017):
      c.execute("INSERT INTO energy VALUES (:
    CountryName, :IndicatorCode, :Year, :Value)", {
      'CountryName': row[0],
      'IndicatorCode': row[3],
      'Year': year,
      'Value': float_or_none( row[headerid[str(year)]]
    )
      })
```

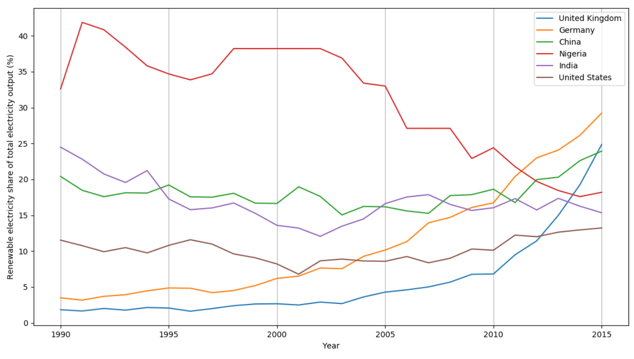
Look at the CSV file to make sure you understand what's happening here (avoiding stuff like this is a great reason to use libraries like `pandas`!) Once we've finished adding rows we have to commit the changes to the database with

```
conn.commit()
```

Exercise 1: Create the database by following the instructions above.

Exercise 2: Write a SQL query to find the countries with the highest renewable electricity share of total electricity output in 2015. Do these numbers seem reliable?

Exercise 3: Plot a time series of the indicator `4.1_SHARE.RE.IN.ELECTRICITY` for a number of countries of your choice



Streaming Algorithms

Simple operations can become very difficult when we have big data. Think about counting how many unique ids are in a file or database of many terabytes. Lots of common algorithms require loading all the data into main memory and if this is not possible we're in trouble. We can make use of the **Data Stream Model**. This means data arrives one element at a time and we have to process each element (or a small subset of the previous history) as we get it. Consider the code below:

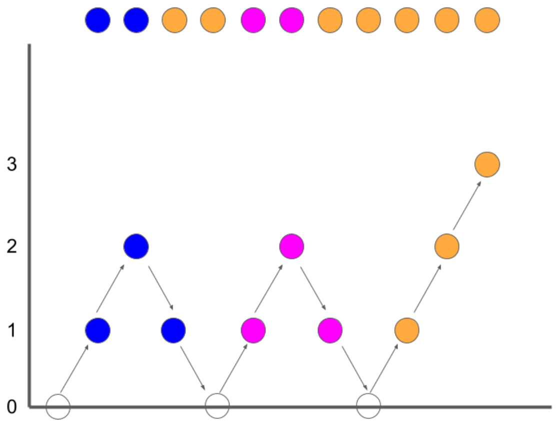
```
import numpy as np
data = np.genfromtxt('BIG_FILE.csv', delimiter=',')
print( len(data) )
```

Exercise 4: What happens in the code above when `bigfile.csv` is one terabyte? How can we accomplish the same thing using $O(1)$ memory? *If you don't know what $O(1)$ means, have a look at the wiki page for big O notation.*

Boyer–Moore majority vote algorithm

Exercise 5: Given a list of items, say `[A, A, B, B, C, A, C, A, A]` write code that finds the most common element.

The most obvious approach is a dictionary. But what if 1. the data is massive and 2. arrives as a stream? The worst case for the dictionary is that we have to store all the values (or at least some large percentage of them). There's no way around this, but there is a clever trick to find the elements that occur at least X% of the time, which is often good enough. The special case of X = 50 is called the **Boyer Moore majority vote algorithm**. The figure below gives an idea of how it works



We start off with no winner. The first element is blue so we store its label and its count is 1. The next element is also blue, we increment the count by 1. The third element is pink. Since pink != blue we decrease the count of blue by one. Continuing in this manner we end on orange.

If there is an element which occurs more than n/2 times, the Boyer Moore algorithm will return it. This type of element is called a **majority element**. Note however, if there is not a majority element, the algorithm may still return something, this means a second pass is necessary to very if that the candidate is in fact a majority element.

The code could be implemented like so

```
elem = None
count = 0;
for d in read(data):
    if count == 0:
        elem = d;
        count = 1;
    else:
        if elem == d:
            count += 1
        else:
            count -= 1
```

Exercise 6: Implement the Boyer Moore algorithm as well as a second loop to check the candidate element.

While this may seem like it has limited use it is fairly straightforward to extend it to find the elements which have a frequency at least n/k for any k. One often cited application of this approach is to detected DDOS attacks. It allows switches (which have very limited memory) to keep track of particular IP addresses and if they are making up a large proportion of the through traffic that can indicate something nefarious is occurring.

Hyper Log Log

Exercise 7: Write (simple!) code to find the number of distinct elements in a list. (This is called the **count-distinct** problem).

The obvious solution again works fine for small data but fails with truly big data. It turns out there is a rather interesting algorithm that we can use to approximate the answer very accurately.

A **hash function** is hopefully something you've come across before. It turns any input into a string of random bits. The probability of seeing a run of k zeros in the bit representation of the hash is $\sim \frac{1}{2^k}$. The algorithm hinges on inverting this idea - if we see a run of k zeros we have probably had to check at least 2^k hash values. So an estimate of the number of elements in the set is

$$N = 2^k$$

Where k is the length of the longest run of zeros observed in any of the hashed values. **This is a terrible estimate!**

The HyperLogLog algorithm uses a number of tricks to make this more accurate

1. Instead of 1 estimate it makes m estimates, using the first few bits of the hashed value to decide on a bucket, then estimating N in each bucket and averaging.
2. Throwing away $\sim 30\%$ of the buckets with the largest values
3. Use the geometric mean, $(x_1 x_2 x_3 \dots x_n)^{1/n}$, instead of the usual mean.
4. Figuring out the right multiplicative factor to correct for bias caused by hash collisions.

The exact details are beyond the scope of this class, but you should know the final answer is accurate with an error of $1.04/\sqrt{m}$, where m is the number of buckets.
Exercise 8: HyperLogLog is often explained by the analogy of going around asking people for the last few digits of their phone number. Write code to generate random 3 digit numbers until 000 is generated. Let the number of numbers generated before stopping be n. By running the code a large number of times and assuming phone numbers are random, how many people do you have to talk to before meeting someone whose phone number ends in 000?

Reservoir Sampling

Exercise 9: Write an algorithm to obtain a random sample of k elements from a data collection.

In some cases a simple approach (just take the first k elements!) can work. However, imagine trying to get a random sample of customers from a big database, if you took k of them in a row they might all share some common features (e.g. joined at the same time) which might not be a good sample of the data. A random sample means for n elements, each element should have a probability of $1/n$ of being chosen. Or, put in another wayz, there is a probability k/n for an element to appear in the sample. There's a neat trick to do this in one pass.

```
reservoir = []
k = 100 #number of samples
for i,d in enumerate( read(data) ):
    if i < k:
        reservoir.append(d);
    else:
        j = random.randint(0,i);
        if j < k:
            reservoir[j] = d;
```

How does this work? We insert item $i > k$ with probability $\frac{k}{i}$. Since there are k items in the reservoir, the probability of replacing any one of them at step i is

$$P(replace_i) = \frac{1}{k}\frac{k}{i} = \frac{1}{i}$$

Therefore the probability of staying in the reservoir is

$$P(stay_i) = 1 - P(replace_i) = 1 - \frac{1}{i}$$

To survive in the reservoir until the end the i^{th} element has to get in the reservoir (with probability k/i) and survive $n - (i + 1)$ chances to remove it. These are all independent events so we can multiply up the probabilities.

$$\begin{aligned} P(survive_i) &= \frac{k}{i}\left(1 - \frac{1}{i+1}\right)\left(1 - \frac{1}{i+2}\right)\cdots\left(1 - \frac{1}{n}\right) \\ &= \frac{k}{n} \end{aligned}$$

Which is exactly what we want from a uniform sample! Again there are many improvements and variations of this algorithm, the important thing is to know something like this exists!

Exercise 10: If you like algebra, prove that $P(survive_i) = \frac{k}{n}$