Using Probabilistic Data Structures to Build Real-Time Monitoring Dashboards

7th Jul, 2017

BY RAHUL

As a provider of Competitive Intelligence as a Service to retailers and consumer brands, DataWeave aggregates and analyses hundreds of millions of products' data from the Web each day. Once aggregated, this data is fed into a complex process of extraction, transformation, machine learning and analyses. These operations are performed on a consistent basis to provide our customers with easily consumable and actionable insights in near real-time.

However, as we keep increasing our coverage of products, there's more stress on our already resource-hungry infrastructure. As a result, we're always on the lookout to improve our data processing efficiency, and ensure the most judicious use of our resources.

One of the ways we achieve this is by making sure only newly acquired data points go through a large part of our data transformation and machine-learning process. This requires us to be able to de-duplicate the acquired data points from the ones already present in our historical datastore, to optimize our operations. By doing this, we significantly reduce complexity and cost, and at the same time, improve our efficiency.

Identifying newly acquired data points, though, is a challenge in itself.

For smaller data-sets, consisting of about 1–2 million data points, a common way of identifying newly acquired data is by indexing the data in a database, and then checking if an element is present or not. This method, however, fails

at the scale that we operate in (hundred million or more data points each day). Also, the associated costs could overrun in the blink of an eye.

On the other hand, our approach is, instead of storing the entire set of data points in the database, only a summary of the data is stored. This method is called Sketching — a technique used in Bloom Filters. While the complexity of the operations here is reduced to O(1), there is a slight trade-off with accuracy.

Probabilistic Data Structures — A Deep Dive into Bloom Filters

Bloom Filters are space-efficient data structures that are used to find out if a given element is present in a set or not, and has a constant time and space complexity for performing operations on the data. Since we store only a summary of the data, an error rate is implicit with Bloom Filters.

(Fun Fact: It was conceived by Burton Howard Bloom in 1970)

The user typically has to make two crucial design decisions while initializing a bloom filter:

- **Cardinality**: It is the maximum capacity of a Bloom Filter. A higher cardinality will result in a bigger memory footprint.
- **Accuracy**: The ideal scenario is to have 100% accuracy in an algorithm. However, this would require a space complexity of O(n). An alternative is to decide on a tolerable rate of error. Lower the accuracy, lower is the memory footprint and faster is the algorithm.

Properties of Bloom Filters

- Set operations like Intersection and Union of two or more bloom filters can be performed
- False Positives (element is not present but the Bloom Filter indicates it's present) can occur but never False Negatives (Bloom Filter indicates element is not present but element is present in the set).
- Operations for element lookup can be parallelized.
- If the cardinality of an element is more than the configured cardinality, the error rate increases.

How Do Bloom Filters Work?

A Bloom Filter passes the key through a series of hash functions (or the same hash function through different seeds). Each hash function generates a value and the bit code corresponding to the value is set in the bitmap.

For Example, **Insertion:**

Consider a Bloom Filter with 20 bits and 2 hash functions. Let's insert 'www.dataweave.com' into the bitmap.

Stage 1:

hash_fn_1(www.dataweave.com) sets the following bits:

Stage 2:

In addition to the above bits, hash_fn_2 (www.dataweave.com) sets more bits in the bitmap:

Search:

To check if a value is present in the bitmap, the same hash functions (hash_fn_1 and hash_fn_2) are run on the key. If all the bits corresponding to the hash functions are set, we can safely assume that particular key is present.

It may also be that a different key generates the same bit pattern. In these cases, the results will be false positives.

If we look to reduce the number of false positives, we will need to increase the 'accuracy', which in turn increases the number of hash functions (and also the number of bits). One can fiddle with the code snippet given in the next section to fine tune the Bloom Filter.

Deletion:

Deletion is not supported as it may result in False Negatives.

Provisioning a Bloom Filter

So, how do we know the number of bits and hash functions required for a Bloom Filter? The answer can be found by using the formula given below, where n is the capacity and p is the accuracy.

Code Snippet:

```
from math import log

import math

n = 1000000

p = 0.01

m = math.ceil((-n*log(p)) / (pow(log(2),2)) )
```

```
k = math.floor((m/n) * log(2))

print("Number of bits" + str(int(m)))

print("Memory Footprint" + str(round((m/8)/(1024*1024),2))+ "MB")

print "No of Hash Functions" + str(int(k))
```

Finding the size of the Bloom Filter and the number of hash functions required for a capacity of 1 million, with an accuracy of 99%:

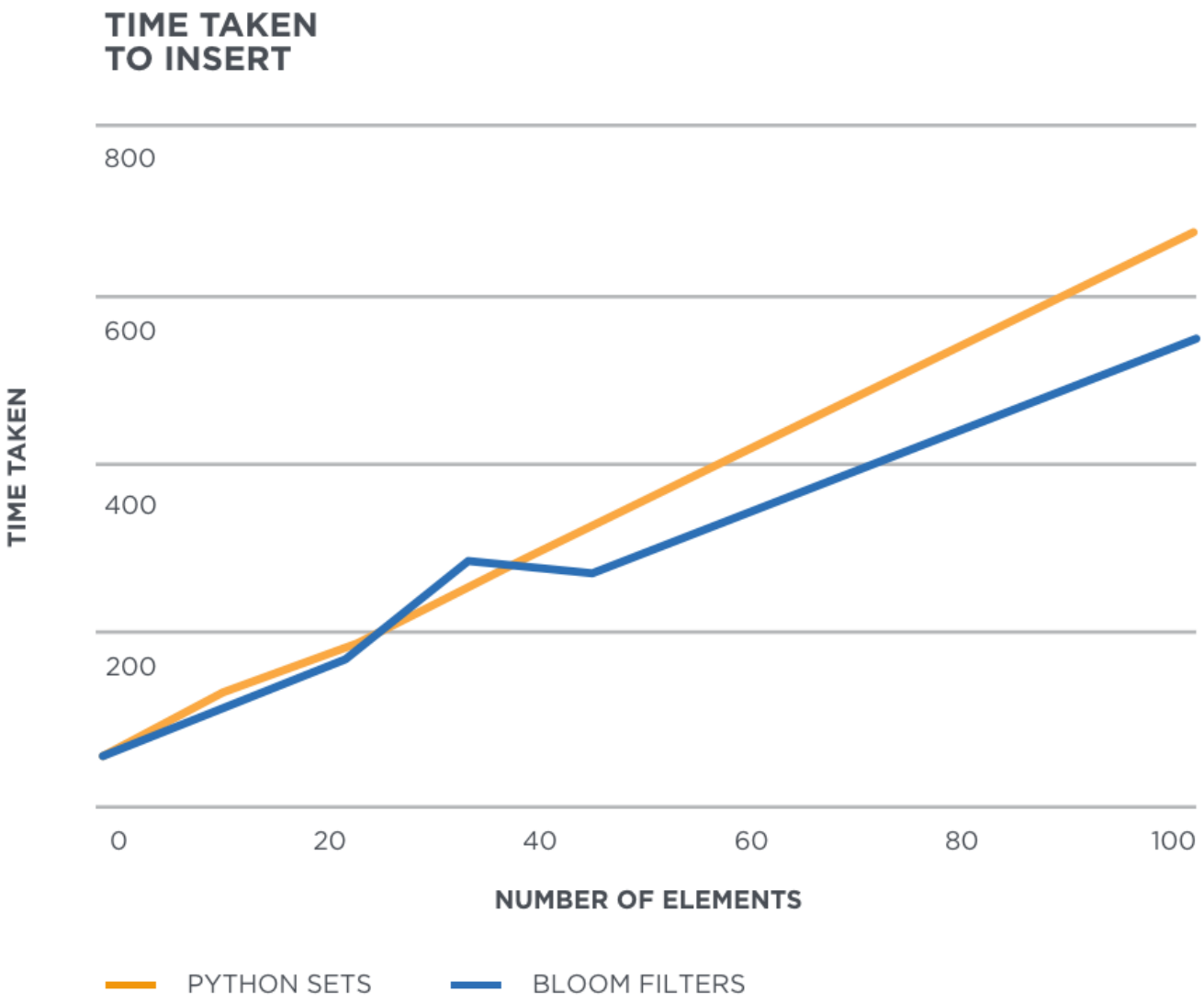
```
Output:

Number of bits: 9585059

Memory Footprint: 1.14 MB

No of Hash Functions: 6
```

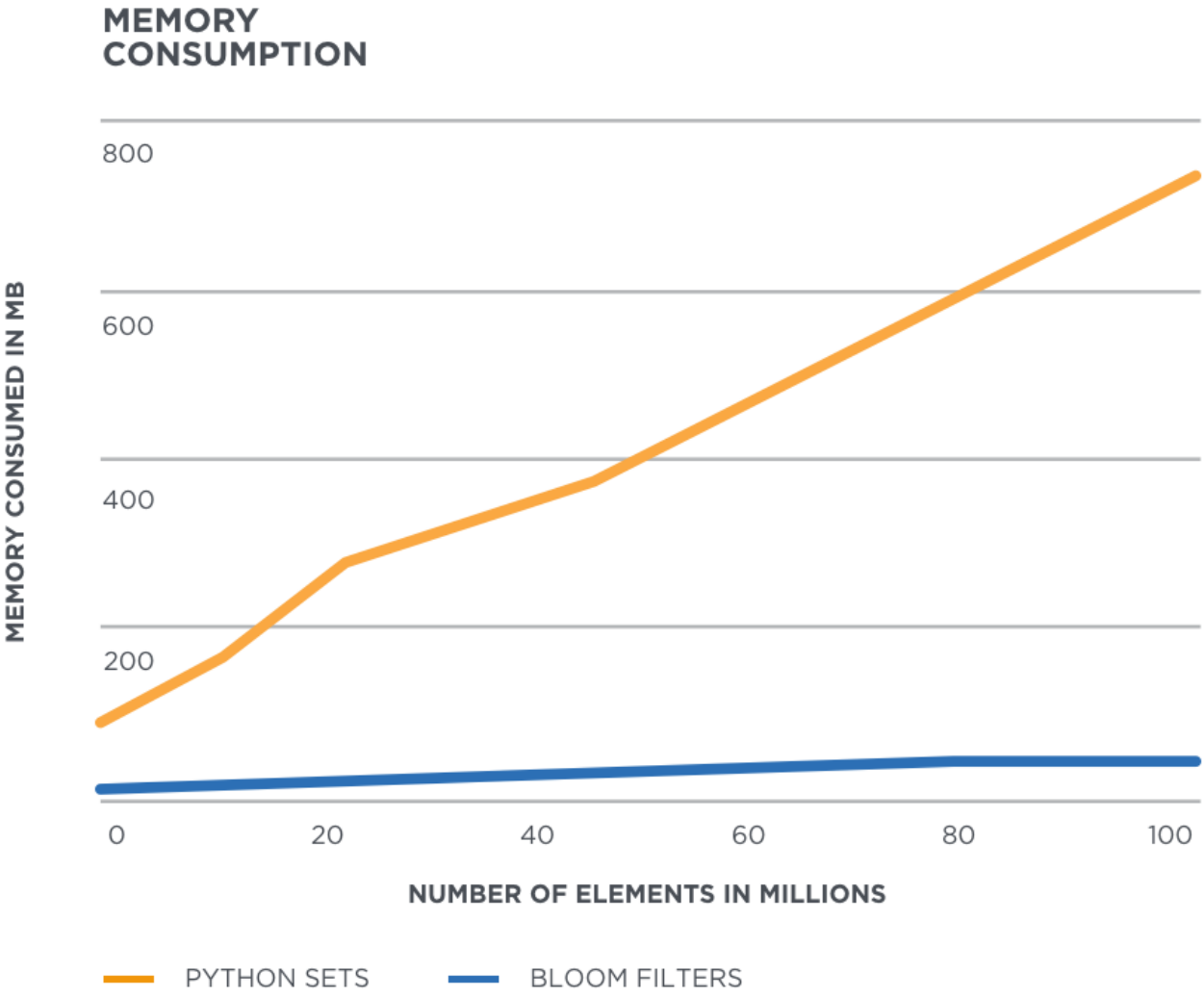
We can find out if an element is present in a set by using just 1.14 MB of memory with a 1% error rate (False Positives).

Benchmark Results:

Some of the data structure types that can be used to perform similar operations include Binary Search Trees, Tries, Bitmaps, Hash Tables, and more. We compared the time and space complexity of Bloom Filters with that of Python Sets (which are implemented using Hash Tables).

- All items had a cardinality of half the number of items. For example, if we were testing for 10 million unique entries, the cardinality was 5 million.
- Bloom filters had an accuracy of 80%.
- The time taken was comparable to that of Python Sets. The value increases linearly with increase in number of elements.
- What's most interesting here, is the amount of memory used by our program, compared to that of Python Sets.





Some Practical Use Cases

Bloom Filters are extremely useful if one is dealing with very large amount of data and a certain degree of error is acceptable.

Cassandra:

Cassandra is a very popular NoSql datastore, and stores data in physical files called SSTables. Depending on the configurations and insertion rates, there can be multiple SSTables in Cassandra. When a user searches for a particular row, Cassandra has to search through the SSTables to return results — a highly inefficient process. To make matters more complex, a single record can be present in multiple SSTables. Cassandra has to then pick the latest one.

To circumvent these challenges, Cassandra uses Bloom Filters to check if a particular record is present in an SSTable. This greatly reduces response times.

What happens if there are False Positives? Cassandra searches through SSTables in which the data might not reside — a process that's largely acceptable as it only results in higher latency. However, Casandra will never miss an SSTable in which the data is likely to reside.

Here is a snippet of the performance statistics of a Bloom Filter used by Cassandra in a production server. The configured accuracy is 99%, and the table has over 200 million rows!

```
Bloom filter false positives: 0
```

```
Bloom filter false ratio: 0.00000
```

```
Bloom filter space used: 152.85 MB
```

```
Bloom filter off heap memory used: 152.84 MB
```

Chrome browser

Chrome uses Bloom Filters to check if a particular URL is malicious in nature. If it is, a warning is issued to the user.

Bloom Filters at DataWeave:

As mentioned earlier in this article, DataWeave uses Bloom Filters to identify newly acquired data points in the last 7 days or 30 days, from a master-set of all data points harnessed each day. This also helps us optimize our scheduling to avoid duplicate data acquisition procedures.

We maintain a Bloom Filter for a daily capacity of 100 million, with an error rate of 10%. To de-duplicate data-points from all of the data harnessed in the previous 'N' days, we perform a Union operation on Bloom Filters over the last 'N' days, and check if the data point is present in the newly created Union Bloom Filter.

Click here to visit our website and find out about how retailers and consumer brands benefit from using DataWeave's Competitive Intelligence as a Service.

Find what we do interesting? Give us a shout in the comments section below, follow us on **Facebook**, **LinkedIn**, and **Twitter**, or **join us** at DataWeave!

- Rahul Ramesh

Technical Architect at DataWeave, 7th Jul, 2017

DATA ENGINEERING