Noel Welsh

_.underscore

Streaming Algorithms

Goal: Understand some scalable algorithms for handling big data

Big Data?

~~Big Data~~

Just big data

Our Demands

Our Demands

- Ridiculously scalable

Our Demands

- Ridiculously scalable
- Real time

Our Demands

- Ridiculously scalable
- Real time
- Simple to implement

Streaming Algorithms

- Process data in one pass
- Limited computation per data item
- Space usage varies, but typically small

The Price?

Probably approximately correct answers

With high probability the answer is close to the true value

Overview

- Hash functions
- Bloom Filter
- Distinct Values
- Frequent Items
- Beyond

Hash Functions

Streaming algorithms
love hash functions!
Let’s do a quick review

Deterministic

Uniform distribution

Bit values are independent

In Practice?

Use Murmur Hash 3

In Scala

- scala.util.hashing Scala 2.10+
- Google Guava

Bloom Filter

Bloom Filter

- A set. “Have I seen this user before?”
- Uses 5 times less space or better than equivalent hash table
- A chance of false positives

Hash Table

- Standard set data type
- One hash (typically 32-bits) + each element
- Exact answers

Bit Set

index = hash(data) mod m

1	0	0	1	0	1	0	1

Bit Set Properties

- One bit per element
- If hash values collide, we can get false positives. Can believe a value is in the set when it is not
- No false negatives

Bit Set To Bloom Filter

- Use more than one hash function
- Allows interesting tradeoffs between space usage and false positive rate

Bloom Filter

$$index_1 = hash_1(data) \bmod n$$

$$index_2 = hash_2(data) \bmod n$$

1	0	0	1	0	1	0	1

Tradeoffs

- More hash functions increase probability of finding a zero bit
- More hash functions increase space use per element

Bloom Filter Maths

- Bit array has size *m*
- Insert *n* elements
- Use *k* hash functions

$$Pr(\mathrm{bit}_i = 0) = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{\frac{-kn}{m}}$$

False Positive Rate

$$Pr(\text{false positive}) = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k}$$

$$\approx \left(1 - e^{\frac{-kn}{m}}\right)^{k}$$

- Can solve to optimise *k* given *m* and *n*
- More usefully solve to optimise *m* given *k* and *n* and false positive rate

Optimal *k*

$$\text{optimal } k = \frac{m}{n} \ln 2$$

- Assume *m* and *n* are fixed
- Solve by taking derivative of previous equation

Optimal *m*

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

- *p* is the false positive rate
- Assume optimal value of *k* from previous equation

Example

$$m = -\frac{2 \times 10^6 \ln 0.05}{(\ln 2)^2}$$

$$\approx 12470449$$

- 2 million distinct elements
- Desired p is 0.05
- 6.2 bits per element

Bloom Filter Tricks

- Union is OR
- Intersection is AND
- Trivial to parallelise or distribute

Practical Issues

- Use Murmur Hash 3
- To generate *k* hashes, hash twice with different seeds, then linearly interpolate *k* values between these hashes

Distinct Values

Distinct Values

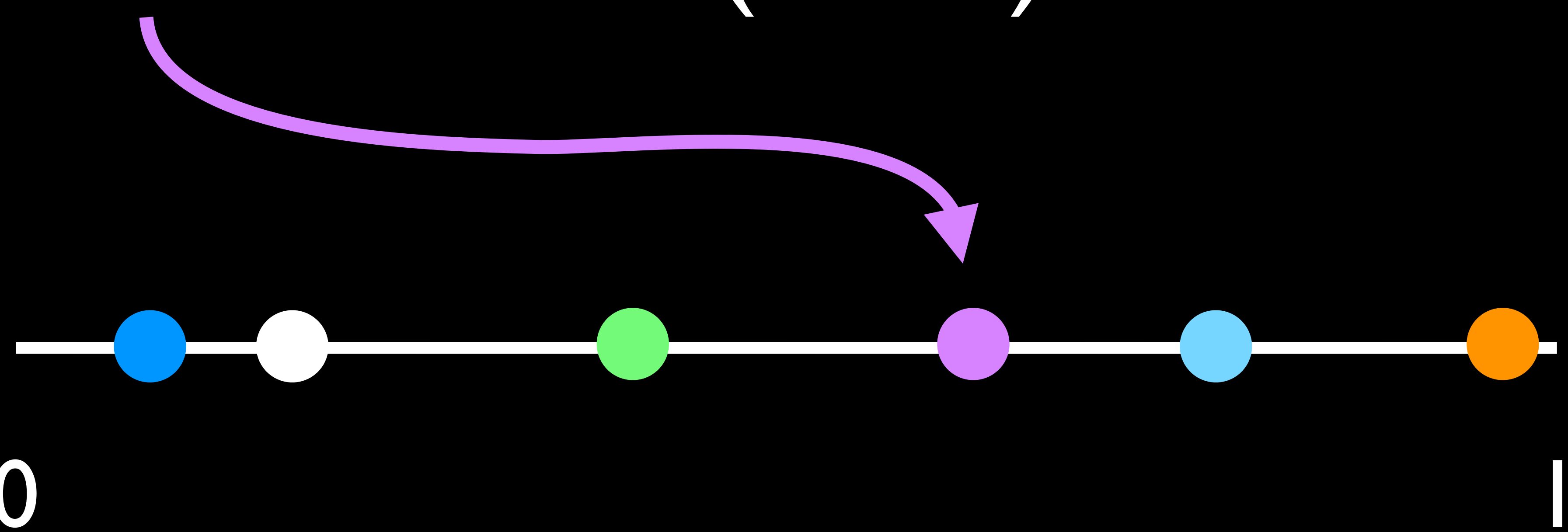
- Count the size of a set. "How many users arrived from LSUG?"
- Can answer with a Bloom filter (and auxiliary counter) but can be *vastly* more space efficient

Many Roads

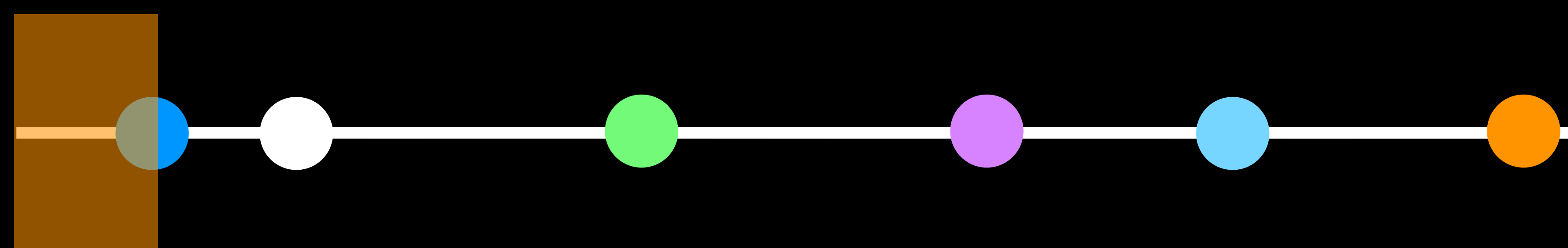
- A lot of research has been done
- Flajolet-Martin sketches (LogLog and HyperLogLog) are popular
- Optimal (but complex) algorithm published in 2010

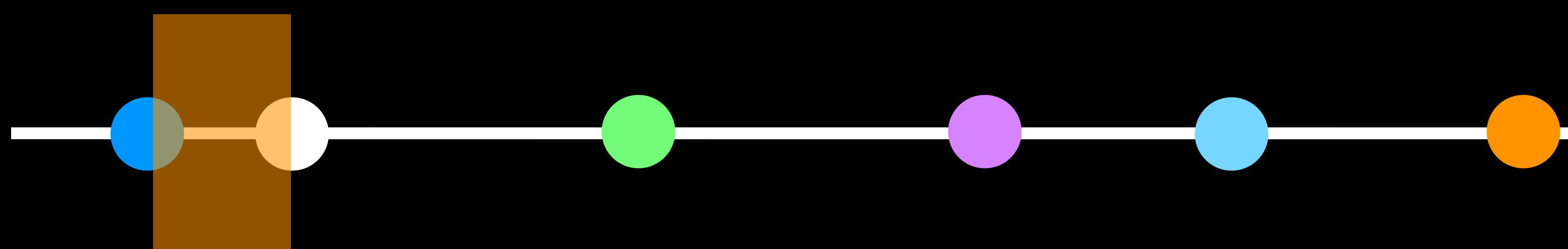
k-Minimum Values

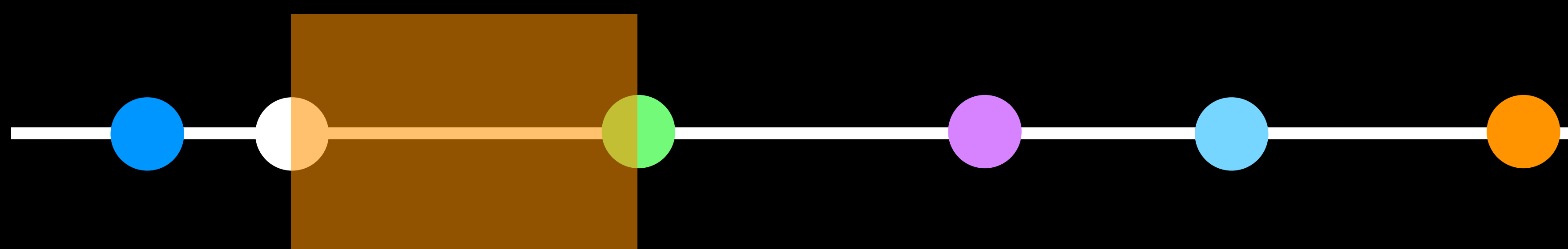
index = hash(data) / maxHash

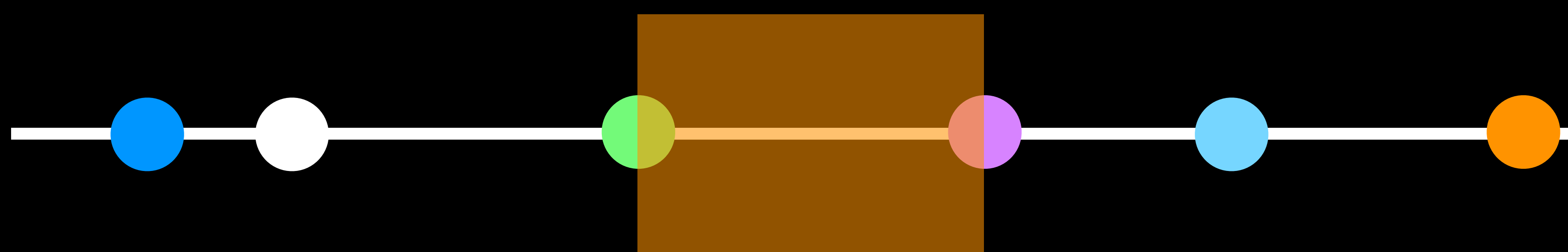


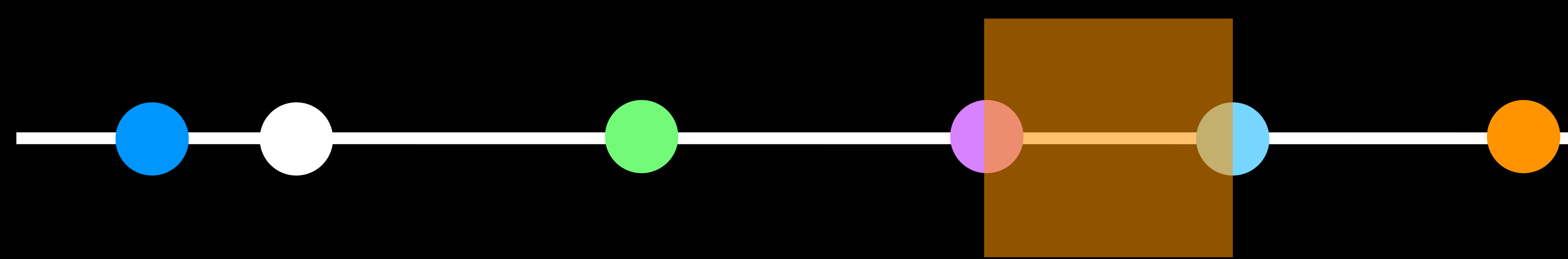
Average distance between elements inversely proportional to cardinality

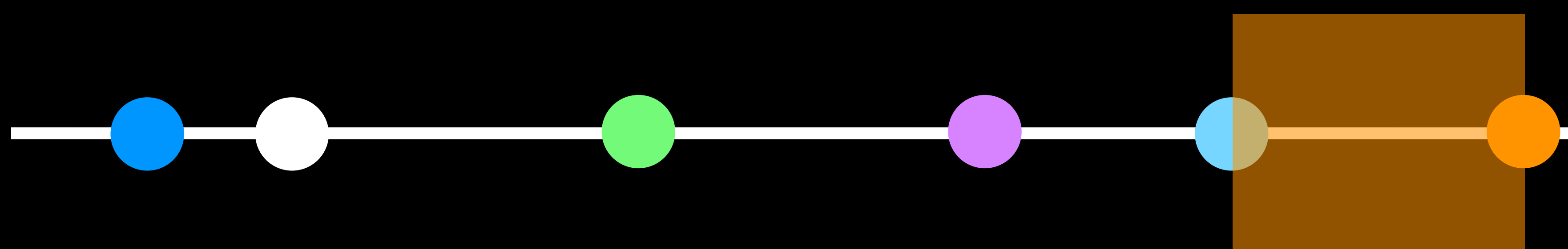


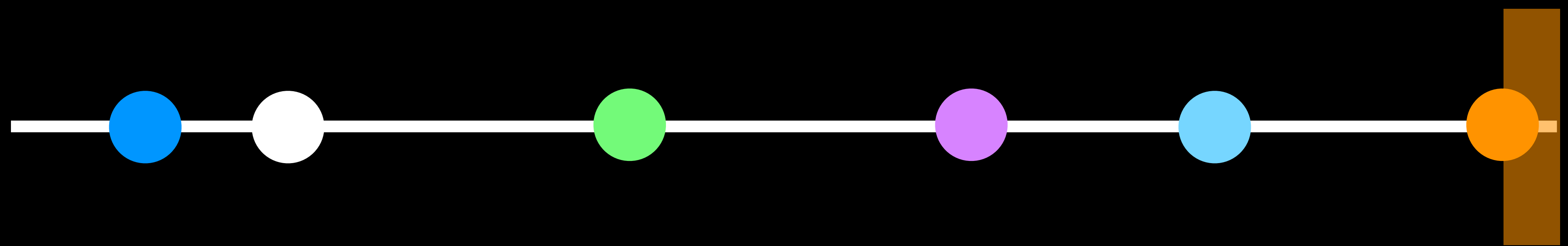










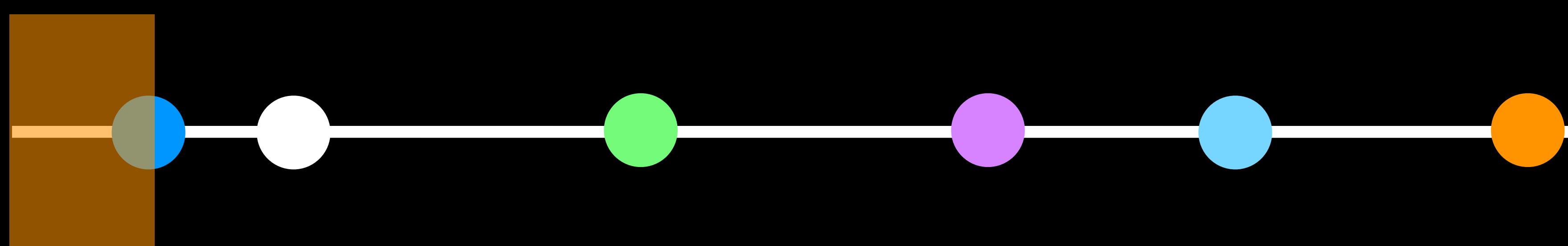


Can’t store all these distances

Big Idea

- Store minimum value. This gives us *one* distance
- Estimate size of set

$$|S| = \frac{1}{\text{minimum}}$$

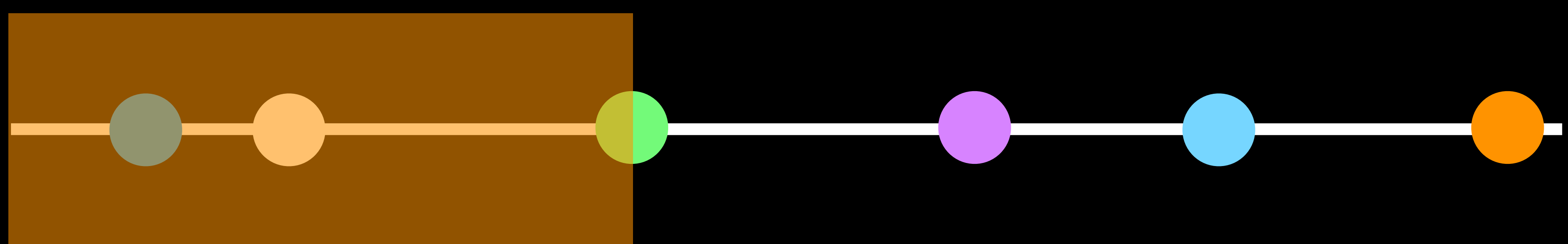


Very noisy!

Refinement

- Store *k* minimum values
- Estimate cardinality as

$$|S| = \frac{k-1}{\text{largest value stored}}$$



$k = 3$

Error Rate

$$\mathbb{E}\left[\frac{|S|_{est} - |S|}{|S|}\right] \approx \sqrt{\frac{2}{\pi(k-2)}}$$

- Independent of size of set
- See papers for other error bounds

Example

- Storing k = 1024 values (typically 4K) gives expected error of 2.5%

k-MV Tricks

- Set union is just the *k* minimum values from the union of the two sets
- Set intersection from Jaccard coefficient
- Set difference if we add counters to each element we store

Frequent Items

Frequent Items

- Find and count occurrence of most frequent items in set. "Who are our most active users?"
- Many approaches

Space Saver

- Store *k* tuples of (item, count)
- Observe item
    - If it's in our list, increment the count
    - Otherwise remove the least frequent item and replace with this one, keeping the count

That's it!

Properties

- Deterministic
- Uses O(*k*) space
- Constant time updates
- Error depends on data distribution

More

Any more for any more?

Code

- Clearspring's stream-lib implements most of the algorithms discussed (and more) in Java.
  https://github.com/clearspring/stream-lib

- Some toy implementations in Scala
  https://github.com/noelwelsh/fleet

Writing

- Lots of blog posts, tutorials, etc. Ask Google
- Alex Smola's course is a good overview http://alex.smola.org/teaching/berkeley2012/streams.html
- *k*-Minimum Values is in http://www.mpi-inf.mpg.de/~rgemulla/publications/beyer07distinct.pdf

Other Algorithms

- We’ve only touched the surface
- Quantiles, clustering, graph properties, etc.
- Online learning is an area I’m excited about. Goes beyond summarising data to taking actions.

Me

- Slides will be on noelwelsh.com
- noel@underscoreconsulting.com
- @noelwelsh