

# Big(ish) Data

Working the Middle Ground Between RAM  
and a Cluster

Riot Games Oct 5 2015

Hi, I'm Kyle Burton currently the VP of Software Development at Relay Network. I've been doing what I love for about 20 years now: writing and learning about software. I love functional programming and Clojure...you may be exposed to trace amounts of Lisp during this presentation.

Please ask questions as we go along...

# When RAM is no longer Enough

## Random Sampling Duplicate Detection

This talk is about a couple of techniques that I've learned that stretch what can be done in a single process and on a single machine. They help you use a fixed amount of memory to achieve the same outcomes without pulling an entire data set into memory. I'm going to cover these two problems, and possibly a third depending on time.

# Mission: Probable

## 20k Random Sample

## Population size: ~400 Million

My team received a data set and wanted to get a sense for how accurate it was. We wanted to pull a manageably sized random sample and do some detailed verification and use that to estimate how good the whole thing was. So let's see...what comes to mind?

# I know I'll use the Database...

```
SELECT *  
  FROM some_huge_table  
 ORDER BY RANDOM()  
LIMIT 20000;
```

This works, though...it's a lot of work for the database. It will have to generate a random number for every row, then sort the data — just to discard most of it.

Hrm...what other tricks are in my bag?



So, your databases' query planner told you this will take somewhere between the next few days and the heat death of the universe...What else have we got?

# GNU Tools

```
$ sort -r in.tab > out.tab
```

```
$ head -n 20000 out.tab
```

Oh, the gnu tools, they're well engineered and perform fantastically. This works, though it doubles disk usage, and just like with the Database it takes a lot of I/O just to discard the majority of the data set.

You can do this in a single pipeline. If the data set is larger than ram it is just going to spill out to disk anyway (sort uses on disk temp files for large inputs).

Ok, what else can we do?

# Flip a Coin?

We could iterate through the data set, generate a random number for each element and if it's less than 20k out of 400M, select that element. This will let us do one pass over the data, though we will be over or under our sample size by some small amount. If we have random access to the elements, we could generate random indexes into the 400M and select elements that way. We'd have to be a little careful not to select the same record more than once.

Can we refine the 'flip a coin' approach? Should we choose random indexes into the set until we hit your sample size? Sure, sure, that's sampling with replacement though, which may result in duplicates.

Can we get sampling without replacement and hit our sample size exactly?

Was that our last hope? No, there is another...

# Stream Sampling

$$\text{Pr}(e) = N / M$$

**N** = Remaining Sample Size

**M** = Remaining Population Size

Stream sampling: if we know the population size a priori, we can get our sample in a single pass. We set the probability of selecting an element as the remaining sample size divided by the remaining population size. Then for each element of the population, we flip the coin. If we're under the probability, we take it, otherwise we move on...



# Stream Sampling

Decrement  $N$  when a sample is taken

Decrement  $M$  for every element

Each time we process an element, we decrement the remaining population size. Each time we select an element, we decrement the remaining sample size.

# Stream Sampling

$$N = 20,000$$

$$M = 400,000,000$$

$$\text{Pr}(e) = 0.00005$$

This is what the numbers looked like for the data I was working with. For the first element we have a probability of one in twenty thousand.

# Stream Sampling

▶ 1	N=2	M=7	Pr(e)=0.285	MISS
▶ 2	N=2	M=6	Pr(e)=0.333	HIT
▶ 3	N=1	M=5	Pr(e)=0.20	MISS
▶ 4	N=1	M=4	Pr(e)=0.25	MISS
▶ 5	N=1	M=3	Pr(e)=0.33	MISS
▶ 6	N=1	M=2	Pr(e)=0.50	MISS
▶ 7	N=1	M=1	Pr(e)=1.0	HIT [DONE]

This is an example of what the process looks like. Here we've got 7 items and we're looking for a sampling of 2 of them. You can see for the first item, we've got 2/7 or a 28 and a half a percent chance of selecting the first element. For the example I've simulated a miss, so at the second element, we have a 2 out of 6 or 33% chance of selecting the item. What I wanted to show here is that as your random number generator results in more misses, your probability of selecting an item goes up — guaranteeing that you'll get your desired sample size.

# Stream Sampling

```
(with-open [rdr (io/reader "phones.txt")]  
  (random-sample-seq  
    (line-seq rdr)  
    400000000           ; population size  
    20000))            ; sample size
```

I've wrapped this up into a nice Clojure library. If you were a Clojure developer you might get excited by this slide. For the python devs, try to focus on the indentation as it carries similar meaning. For the Java devs...well I'm sorry this probably just looks weird :)

- 
- Sample in One Pass
  - Disk IO is 1x
  - $O(\text{reasonable})$

HELLO  
my name is

*Win*

Streaming sampling was a win, the approach was efficient enough that I was able to test my code and I was able to process my data set in reasonable amounts of time. Certainly less time than the database...

# Related Algorithm

## Reservoir Sampling

for when you don't know your  
population size

This one is also worth looking at. It works well when you don't know the actual population size ahead of time. Reservoir Sampling assumes the sample set will fit into memory, where Stream Sampling does not.

**Questions?**  
**(time?)**

# Next Lurking Issue?

We've talked about taking random samples for doing quality checks and for finding duplicated values...what's the next all-too-common thing with data sets?





Yes, default values. Some programmers...maybe it's business people..think that having some value, any value is better than having nothing. Of course some programmers are wrong...

# aka “Dummy”

- ‘NULL’ / ‘N/A’
- (610) 555-1212
- na@na.com
- 123 Main St.
- John Q. Public

I’m sure you’ve all seen how these manifest themselves. Sometimes they creep in from test code, sometimes it’s to avoid null references. When you’re dealing with millions and millions of records, they can be a pain to find when they’re not in a majority.



So how do you find a few repeated values in a data set where you expect high cardinality? How do you identify the few needles lying in your haystack?

# Naive Counting

```
(defn find-dupes-naive [inp-seq]
  (reduce
    (fn [counts item]
      (assoc
        counts
        item
        (inc (get counts item 0))))
    {}
    inp-seq))
```

You simply count the number of times you've seen every item...unless you have a really large data set, because then...



...we run out of RAM.

# GNU Tools

```
cut -f2 | \  
sort | \  
uniq -c | \  
grep -v '  | ' sort -nr > counts.txt
```

Of course we can reach for the gnu toolchain again... This will also often be faster than doing it in a database...but still costs unnecessary time, sorting large data sets is expensive as it often spills to disk and did I mention it takes up a lot of disk space?

# Is This The Only Way?

I used to think this was the only way to do it, then I read about...



# Bloom Filters

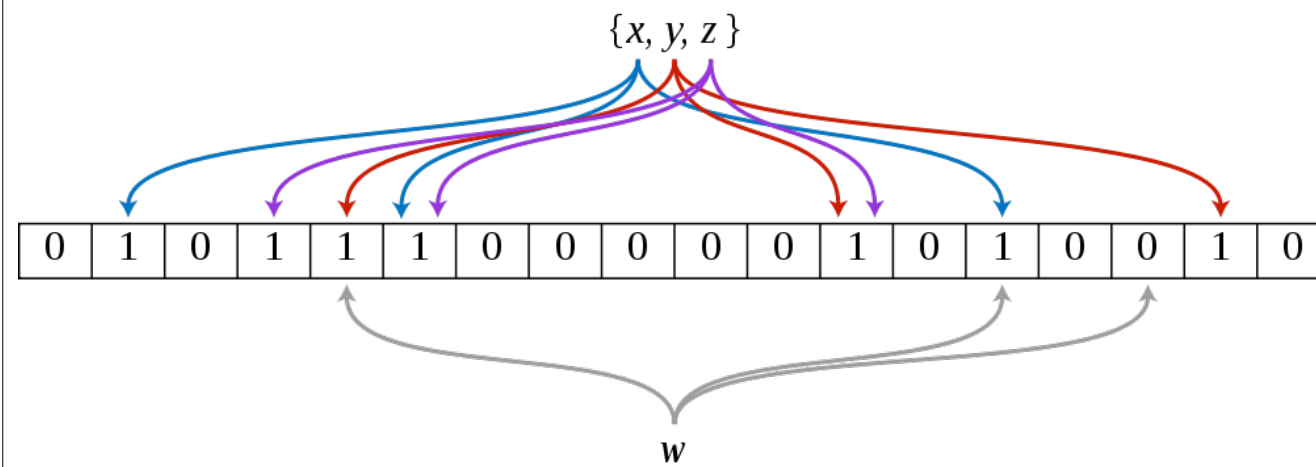


“A Bloom filter is a space-efficient probabilistic data structure, conceived by Burton Howard Bloom in 1970” so sayth Wikipedia.

Bloom filters act like a set, though they make a very crucial trade off — they trade the probability for a false positive for a constant amount of memory. So it may say that an element is in the set when it is in fact not, but it will never say an element is not in the set when it actually is. False Positives but not false Negatives.



# Probabilistic Set



A Bloom filter is a large array of bits. It uses a set of hashing functions. For each element, here  $x$ ,  $y$  and  $z$ , you compute the hashes represented by the colored lines. Each hash's output sets a bit in the array. To test if an element, ' $w$ ' for example, is in the set, you run the hashes and see if all of those bits are on. In this example they aren't, so we know ' $w$ ' isn't in the bloom filter.

# There's Some Math...

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

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

$$\frac{m}{n} \ln 2 \approx \frac{9m}{13n} \approx 0.7 \frac{m}{n},$$

$$\left(1 - e^{-k(n+0.5)/(m-1)}\right)^k.$$

I won't claim I can derive any of these equations. The upper left defines the probability for a false positive where (m) is the number of bits in your array; (k) is the number of hash functions you use and (n) is the number of elements tracked in the filter.

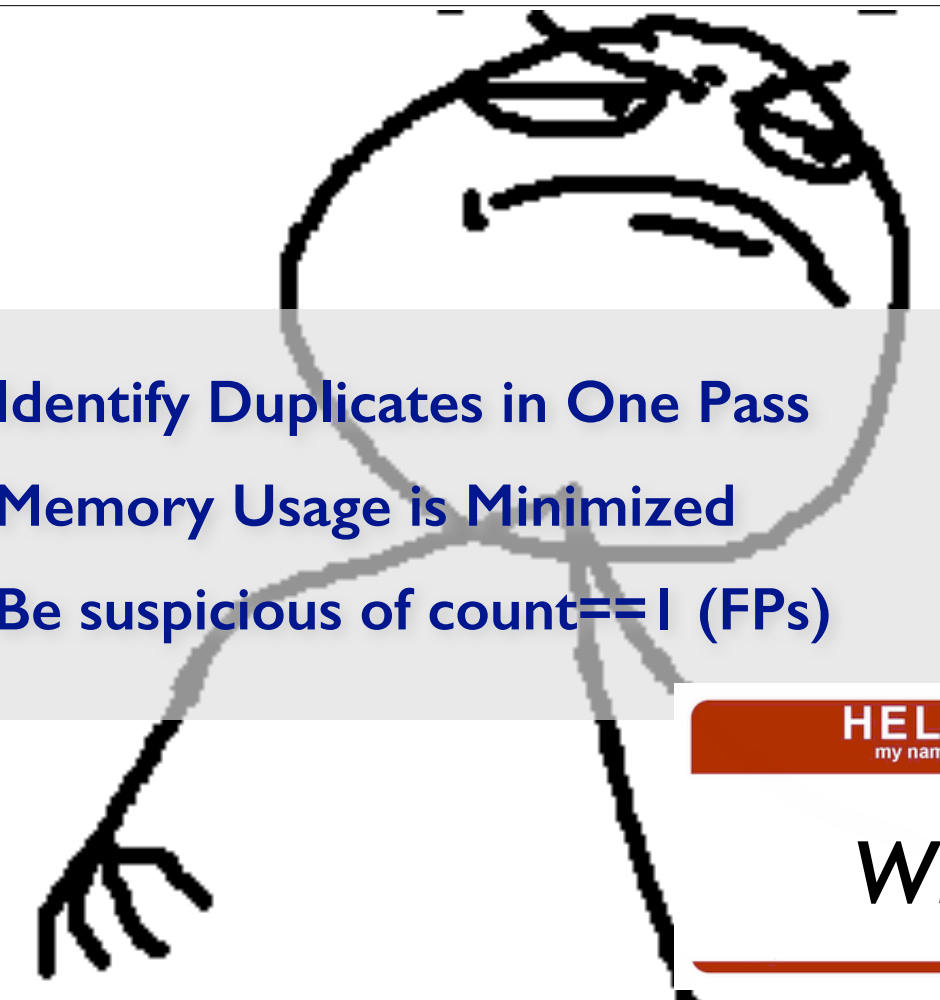
The other three are used to determine the optimal number of hashes (k) and number of bits (m) for a desired false-positive rate. Which is great, we can decide how much memory we're willing to trade for reducing the false positive rate.

# It's a Bloomin' Dupe

```
(defn find-dupes [inp-seq psize fp-rate]
  (let [flt (bloom/make-optimal-filter psize fp-rate)]
    (reduce
      (fn [counts item]
        (if-not (bloom/include? flt item)
          (do
            (bloom/add! flt item)
            counts)
          (assoc counts
            item
            (inc (get res item 0)))))
      {}
      inp-seq)))
```

For Clojure, I've wrapped this up into a library. We're doing two things in this code: first we're populating the Bloom Filter as we process the sequence; the second thing we're doing is counting the items that the Bloom Filter says seen before. This way we're only tracking counts for the items we think are duplicates instead of for the entire data set.

We ask the library to make an optimal filter for us, then we accumulate counts of our items into a map – only adding items into the map when the bloom filter indicates we've seen the item before. After processing the sequence, we'll have some elements with an actual count of 1. These may be because there were only 2 items in the data set, or they may be false positives. Remember, here we're only using the filter to identify values that may be duplicates, we're using the map to count them...starting from the second time we've observed the value.

- 
- Identify Duplicates in One Pass
  - Memory Usage is Minimized
  - Be suspicious of count==1 (FPs)

HELLO  
my name is

*Win*

Any values where the count is greater than one are definitely in the data set multiple times, any where the count is exactly one may be in the data set twice or may be a false-positive. Of course once you have this set, you can actually find the duplicates with one more pass over the data, treating the results from the first run as your candidates.

# Related Algorithm

HyperLogLog

probabilistic cardinality estimation

Related to looking for duplicates is estimating the cardinality (or uniqueness) of a data set. If you're hungry for learning more along these lines, HyperLogLog is worth reading about.

Questions?  
(time?)

# Summary Counts

This third (dare I say bonus?) section is less an interesting algorithm and more about not giving up on what you think the right approach is. It's about counting things...

# You Know...

NY 14,735

PA 11,234

NJ 8,907

DE 5,191

Counts of the states in your data set. We typically only count things when they have low cardinality.



Smells Like

Embarrassingly Parallel



# Smells Like: Map / Reduce

I think this is why hadoop was invented...but can we do anything like this on a single machine?

# Can Haz Multi-Core?

Your machine ~~probably~~ has multiple cores

Maybe even multiple CPUs

FP Langs are Supposed to make it easier to leverage these right?

(hint: They do)

# Divide and Conquer!

```
split -l 100000 \  
    phone-nums-with-lfsr-ids.txt \  
    working-dir/inp-  
  
wc -l working-dir/inp-a*  
100000 working-dir/inp-aa  
100000 working-dir/inp-ab  
100000 working-dir/inp-ac  
100000 working-dir/inp-ad  
100000 working-dir/inp-ae  
...
```

Before I get to that, don't forget if you're on a Unix system, the GNU tools come with some things to help you out: namely 'split'

# Divide and Conquer!

```
(defn count-area-codes [inp-seq]
  (reduce (fn [m line]
            (let [phnum      (second (.split line "\t"))
                  [_ area-code] (first (re-seq #"\\((\\d+)\\)" phnum))]
              (assoc m area-code (inc (get m area-code 0)))))
    {}
    inp-seq))

(apply
 merge-with +
 (map (fn [inp-file]
        (count-area-codes (ds/read-lines inp-file)))
      (map str
            (filter #(.isFile %)
                    (.listFiles (java.io.File. "working-dir/"))))))))
```

Then you have a simple work queue...

Up top we have a function that counts area codes from a file

Below we call it on each of the files created by split

On my mac this takes about 43seconds

# Divide and Conquer!

Did you notice the letter ‘p’  
I added right there?

Did you notice the letter 'p'  
I added right there?

```
(defn count-area-codes [inp-file]
  (reduce (fn [acc phnum]
            (let [t (count-area-codes-in-file inp-file phnum)]
              (inc (assoc acc t 0)))))
    {}))

(count-area-codes "input.txt")

;; => {0 1}

(defn count-area-codes [inp-file]
  (apply merge-with +
    (pmap (fn [inp-file]
            (count-area-codes (ds/read-lines inp-file)))
          (map str
                (filter #(= (.isFile %))
                        (.listFiles (java.io.File. "working-dir/"))))))))
```

That little 'p' saved me 16% on my mac – on which this task is most likely I/O bound, not too shabby, I get better results on my workstations and servers...

# Tricked You

Running Split wasn't free, we also wrote out the data set again, doubling our disk usage. Of course I wouldn't have this slide if I wasn't going to show you a way around that :)

# No Need to Split

Process 'blocks' / 'chunks' in parallel

Ensure they're aligned with line (record) separators.

(I wrapped this up into a library for you)



# No Split


```
(reduce
  (fn [res counts]
    (merge-with + res counts))
  (pmap (fn [[start end]]
    (count-area-codes
      (io/read-lines-from-file-segment
        inp-file start end)))
    (partition 2 1
      (io/byte-partitions-at-line-boundaries
        inp-file
        (* 1024 1024))))))
```

This is parallel map with reduce in a single process

We ask for roughly 1 meg chunks, most will be a little larger than 1mb.

We pass map helper fn that returns a seq of lines from that chunk

This runs in about the same time as processing the files we created with gnu split, but w/o needing to pre-split the file.

- 
- Use Your Cores
  - Avoid Unnecessary I/O
  - Avoid Additional Disk Usage

HELLO  
my name is

*Win*

This Slide Left  
Intentionally Blank

That are all the tricks I have in this talk...before I wrap up, are there more questions?

# Clojure

## Sequence Abstraction

### Lazy

### Composable

I had several functions that took seqs, then I experimented with different ways of creating seqs.  
Using take/drop makes it easy to do little profiling runs, which is vital! in determining when to use map vs pmap

# Clojure

## Easy Concurrency

### pmap

lazy means less OOMs; immutability means easier concurrency

# Iterate Faster

Faster Runs let you work out  
kinks quicker

More Iterations Reduce Bugs

# Thank You

@kyleburton

[github.com/kyleburton](https://github.com/kyleburton)