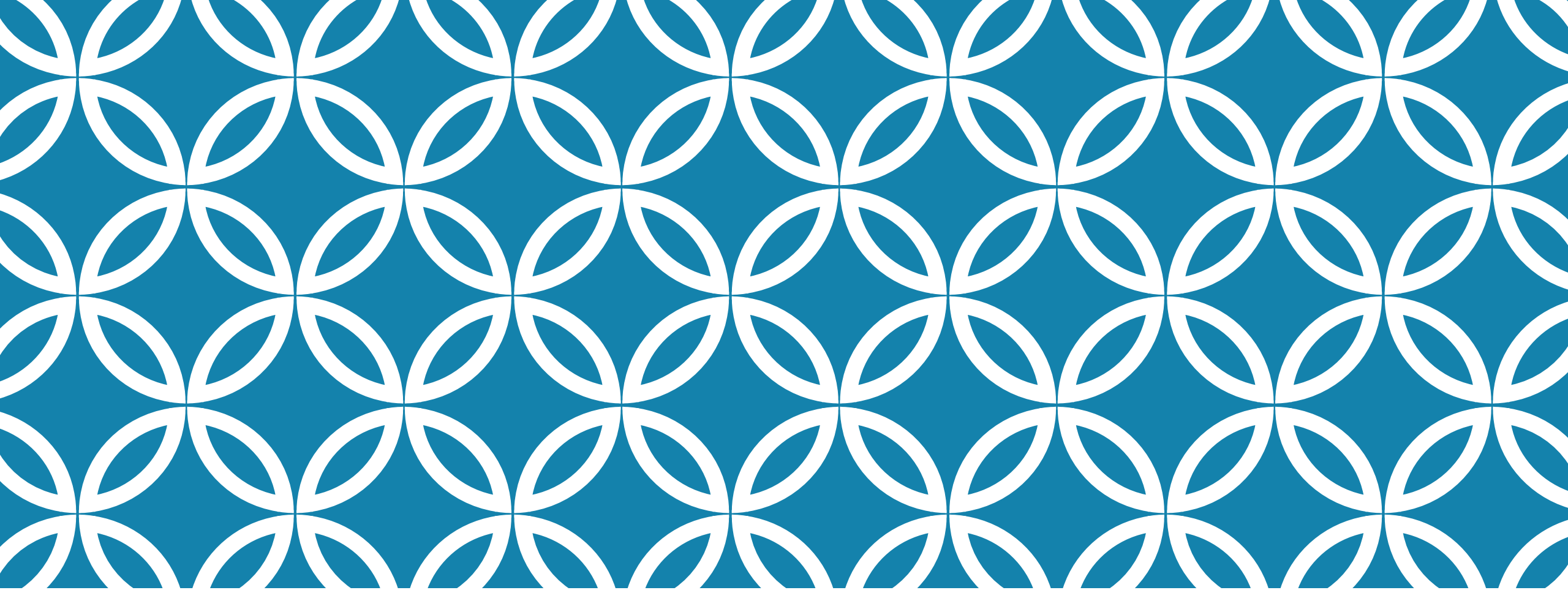


SKETCHING TECHNIQUES FOR REAL-TIME BIG DATA

Stefan Savev
Course@
Data Science Retreat, Berlin

OVERVIEW

- Big Data.
  - Here Big Data == Big Matrix
  - Here data = interaction/co-occurrence data from e-commerce, traffic logs
- Some data analysis questions
  - How many DISTINCT users viewed a page
  - Approximate trends (counts of events over time)
  - Fast linear regression with extremely large number of features (large number of features = all bigrams)
  - Fast dimensionality reduction with the SVD (detect patterns)
- Main Theme
  - Sometimes approximate answers are good enough (When?)
  - We are turning the big data into small data (sketches) on which we can still compute the answers (approximately)
  - Suitable for real time answers and streaming application (real time streaming needs little memory)

WHEN IS APPROXIMATE GOOD ENOUGH?

Aggregate statistics: # users of a website, they change all the time so why focus on complete exactness

Machine Learning – it's already an approximation

Search – it's already based on "fuzzy matching"

Recommendation Systems – it's a blackbox with little guarantees anyway (for example it will work for one customer very well and poorly for another one)

When is approximate NOT appropriate?

Total sales for last year in $

What about projected sales for next year?

INTERESTING TRENDS

Fast data (Typesafe white paper)

- Updating machine learning models as new information arrives.
- Detecting anomalies, faults, performance problems, etc. and taking timely action.
- Aggregating and processing data on arrival for downstream storage and analytics.

Turning big data into little data

- For example to estimate the mean (e.g. mean latency) you don't need to store the whole data
- To estimate a histogram you may need to see all the data, but you don't need to store it

TASKS

- Day 1: Implement memory efficient (approximate) COUNT DISTINCT
- Day 2: Implement Ted Dunning's t-digest together with an anomaly detector
- Day 3: Overview of hashing/sketching/random projection techniques

GOALS

Analysis of the approximate distinct count problem. Memory usage. What are advantages of different approaches (Hadoop, Streaming, exact vs. approximate)

Building an API for this problem

Building a small testing framework for this problem.

Testing mainstream libraries for approximate distinct count (ElasticSearch and Streamlib)

Creating your own simple solution (algorithm) for this problem and testing it.

Rough understanding the HyperLogLog algorithm.

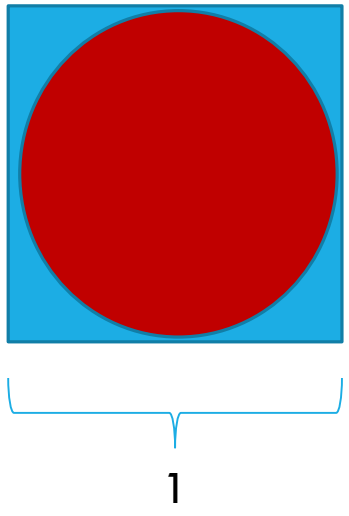
Implementing HyperLogLog without fine tuning.

Understanding where HyperLogLog breaks and why.

Improving upon HyperLogLog with Linear Counting. HyperLog++

Understanding the fine print in HyperLogLog++. Exactly how much memory is used for exactly how much precision.

WARM UP: ESTIMATE π BY THROWING DARTS



- r = 1/2
- Area of circle = $\pi r^2 = \pi/4$
- Area of rectangle = (2r)^2 = 1
- Ratio = Area of circle/Area of square = $\pi/4$
- π = 4*Ratio

On the other hand:

- Ratio = fraction of darts that fall in the circle

(note that all darts fall in the square)

WARM UP: ESTIMATE π BY THROWING DARTS

```
val numDarts = 1000000
val rnd = new Random(4811)
case class Point2D(x: Double, y: Double)
val circleRadius = 0.5
def throwDart(): Point2D = Point2D(…,…)
def inCircle(point: Point2D) = … //test whether the point is in the circle
val numDartsInCircle =
      (1 to numDarts).map(_ => throwDart()).count(inCircle)
def estimate(numDartsInCircle: Int, numTotalDarts: Int): Double =…
println("pi estimate: " + estimate(numDartsInCircle, numDarts))
```

STOCHASTIC AVERAGING EXPLAINED

APPROXIMATE DISTINCT COUNT

Use case:

- An organization has many (e.g. millions) of web pages
- Each page is hit by many (e.g. millions) of users.
- We want to be able for each page to find the number of DISTINCT users that saw the page for a given time interval
- Special case: count all unique visitors on a web site

Other use cases

- Use case at Google: number of unique queries for a given time frame
- Use case in database optimizers: Select good query plans
- Extended use case: range queries, show trend of unique counts
- Fast computation of similarity

POSSIBLE SOLUTIONS

- Naïve solution: use a hash set for each page. Next slide: compute memory
- Brute force solution: store the page-visitor interactions in Hadoop and run offline jobs.
- Use **HyperLogLog++** algorithm: approximate solution

WHO USES HYPERLOGLOG(++)?

- Redis New Data structure: http://antirez.com/news/75
- ElasticSearch: https://www.elastic.co/blog/count-elasticsearch

  (according to ElasticSearch this feature has been requested by many users)
- Amazon: http://www.looker.com/blog/practical-data-science-amazon-announces-hyperloglog
- Facebook in Presto database

DISTINCT COUNT: NAÏVE SOLUTION

- Uses a hash table for each page
- One million web pages
- How much memory is used in (GB?)
- Next slide: assume Zipfean distribution.

DISTINCT COUNT: ZIPFEAN DISTRIBUTION ANALYSIS

Word	Freq	r	Pr(%)	r*Pr
the	2,420,778	1	6.488	0.0649
of	1,045,733	2	2.803	0.0561
to	968,882	3	2.597	0.0779
a	892,429	4	2.392	0.0957
and	865,644	5	2.32	0.116
in	847,825	6	2.272	0.1363
said	504,593	7	1.352	0.0947
for	363,865	8	0.975	0.078
that	347,072	9	0.93	0.0837
was	293,027	10	0.785	0.0785
on	291,947	11	0.783	0.0861
he	250,919	12	0.673	0.0807
is	245,843	13	0.659	0.0857
with	223,846	14	0.6	0.084
at	210,064	15	0.563	0.0845
by	209,586	16	0.562	0.0899
it	195,621	17	0.524	0.0891
from	189,451	18	0.508	0.0914

NumberWordsOccur(n) = V / [n * (n + 1)]
is the number of words that occur n times.
where V is the number of unique words in the collection

Use case: fraction of the words that appear once
Set n = 1, then V/2 is the result,
Half of the words appear only once

- Reference

MEMORY FOR ALL PAGES

Half (1/(1*2)) of the web pages have one unique view

(1/(2*3)) of the pages have 2 distinct users

(1/(3*4)) of the pages have 3 distinct users

And so on.

$1*1/(1*2) + 2/(2*3) + \ldots + n*1/(n*(n+1)) = \log(n)$

Basically:

- Most pages are viewed a few times [small memory to count]
- A few pages a looked at by a lot of users [lots of memory but for a few pages]

DISTINCT COUNT WITH HADOOP

- Input: a collection of (page_id, user_id) tuples
- Goal: for each page compute number of unique user id's
- Simple solution:

```
def mapper(key=page_id, value=user_id):
    emit(page_id, user_id)

def reduce(key=page_id, values_iterator=user_ids_per_page_id):
    num_distinct = disctinct(user_ids_per_page)
    emit(page_id, num_distinct)
```

- Problem: page_ids with largest number of distinct users may cause out-of-memory error in reducer or the corresponding reducer to run very long

DISTINCT COUNT WITH HIVE

SELECT page_id, **COUNT(DISTINCT** user_id) AS num_distinct_ids FROM visits

GROUP BY page_id

Question: how many map reduce jobs does HIVE run for this?

DISTINCT COUNT WITH SCALA

```
val data = Array((page_id, user_id))

data.groupBy({case (page_id, user_id) => page_id}).

map({case (page_id, user_ids) => page_ids, Array.distrinct(user_ids).length})
```

APPROXIMATE COUNTING

```
class ApproximateCounter (precision: PrecisionSpec){
        def add(id: Long): Unit = …
        def distinctCount(): (Double, StdDev) = …
}
```

- T can be long (e.g user id), string (e.g. cookie), tuple (e.g. (user_id, page_id))
- Precision: how to specify it?
  - ElasticSearch uses a number between 0 and 40000. Default is 40000. What does it mean?
  - More later
- Counts are integers (or longs), but some algorithms may return double because we return an approximation. ElasticSearch and StreamLib return a Long.
- Since the result is a random variable, we can return the std. deviation as well
- Approximation Method = Linear Counting, HyperLogLog, HyperLogLog++, etc.

TASK 1: DESIGN/IMPLEMENT THE API

Suggested API:

```
trait ApproximateCounter{
  def add(obj: T): Unit
  def distinctCount(): Double
}
```

TASK 2: IMPLEMENTATION WITH STANDARD HASH TABLE

Create a verifying implementation with Scala/Java HashMap

class NaiveCounter extends ApproximateCounter{

- …

}

TASK 3: CREATE A WRAPPER AROUND STREAM-LIB IMPLEMENTATION

TASK 4: CREATE A WRAPPER AROUND ELASTIC SEARCH IMPLEMENTATION

TASK 5: TEST ON RANDOM DATA

- Create a generator class

```
class ItemGenerator(objectCounts: Array[Int], rnd: Random) {
        def onProcessItem(processor: Int => Unit): Unit = …

}
```

- objectCounts is an array such that objectCount[i] = k means that object i will be generated k times

For example if objectCounts = Array(2,1,4) we can generate a sequence 0,0,1,2,2,2,2 or any permutation of this sequence. The generated items are send to onProcessItem(...) and received via the function processor

Or just a generator class that outputs 1,2,3,….

Or use an iterator

TASK 5:

Create a Tester object. For example

```
class Tester(itemGenerator: ItemGenerator,
                counterImplementations: Vector[ApproximateCounter]) {
   val printWriter = new PrintWriter()
   itemGenerator. onProcessItem(itemId => {
       for(impl <- counterImplementations){
              impl.add(itemId)
              printWriter.printLine (impl. distinctCount().value.toString())
       }
  })
  printWriter.close()
}
```

TASK 6: EVALUATE THE ACCURACY OF THE COUNTERS

The file created from the previous step should contain at least two columns

the first column is for the naïve (exact) distinct count.

the second count is for the approximate distinct count.

We want to plot them against each other (for example in R or Python)

TASK 7: OUR FIRST IMPLEMENTATION

Suppose we are working with long numbers that represent ids.

We map each id to a random number in the interval (0,1).

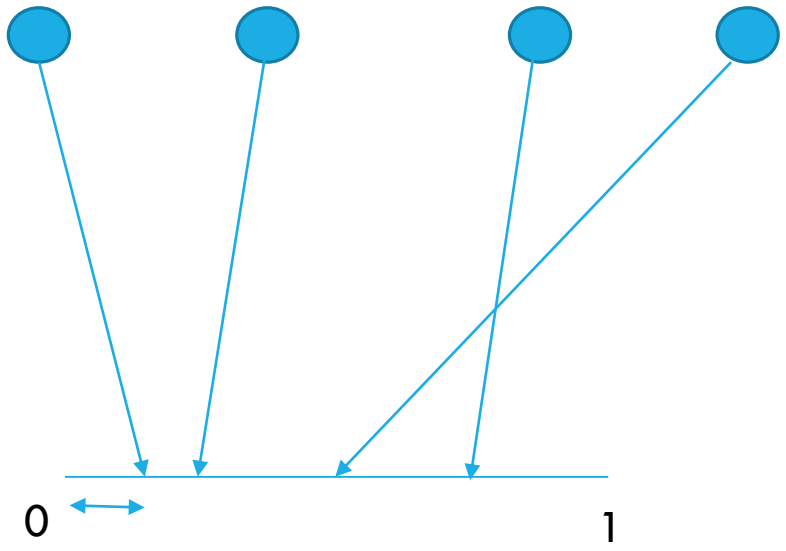
If k unique items are seen, they are mapped to k unique numbers in (0, 1).

Those numbers split the interval into (k + 1) parts. The expected lengths of each part are equal and since they sum to 1, the expected length of any part is 1/(k + 1)

Therefore if we know the smallest number in the interval, we can solve for k

1/(k + 1) = E[first interval length] = smallest number.

Number of distinct objects = 1/smallest number - 1



ONE ALGORITHM

```
counters = Array.ofDim[Double](k).map(_=> 1.0) //initialize to 1

def add(obj): Unit = {
        for(i <- 0 until k){
                hash_code_i = hash(i, obj) //assume output from [0 to MAX_LONG)
                rnd_number = hash_code_i.toDouble/MAX_LONG
                counters(i) = Math.min(counters(i), rnd_number)
        }}}
def getDistinct(): Double = {
        avg = Array.average(counters)
        1.0/avg - 1
}
```

IMPROVEMENT

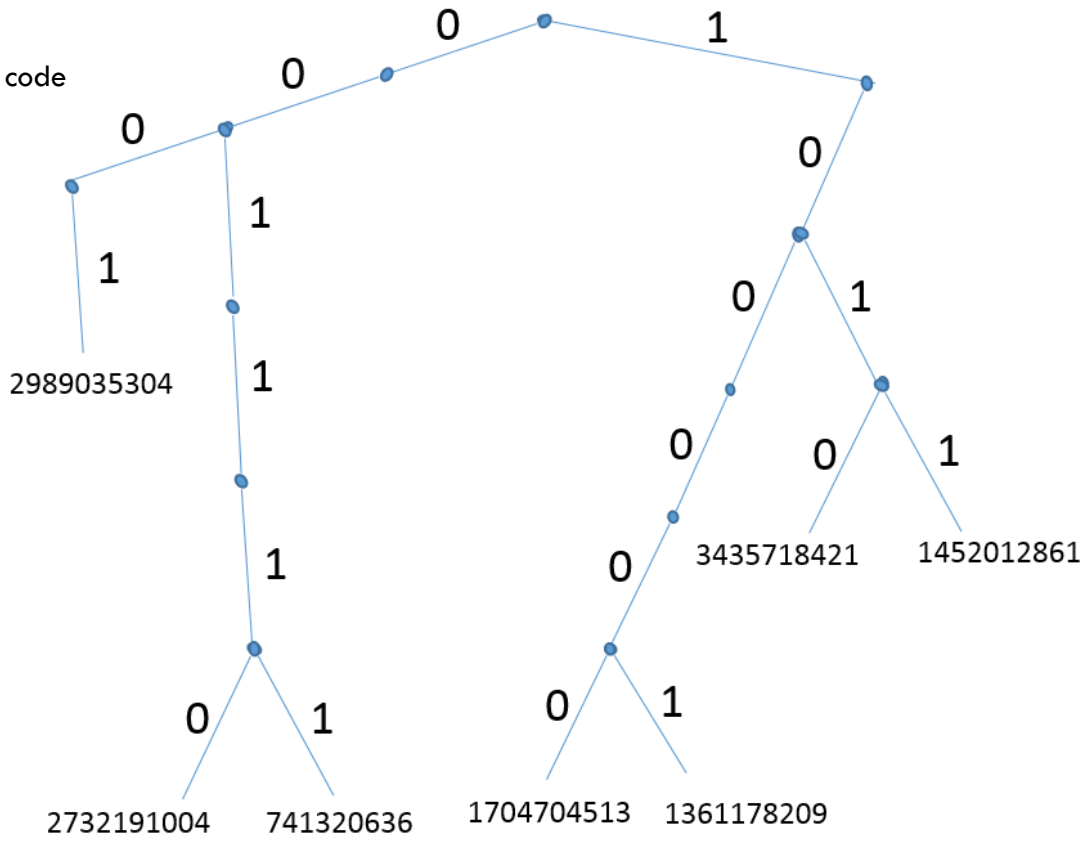
Add more counters
then average

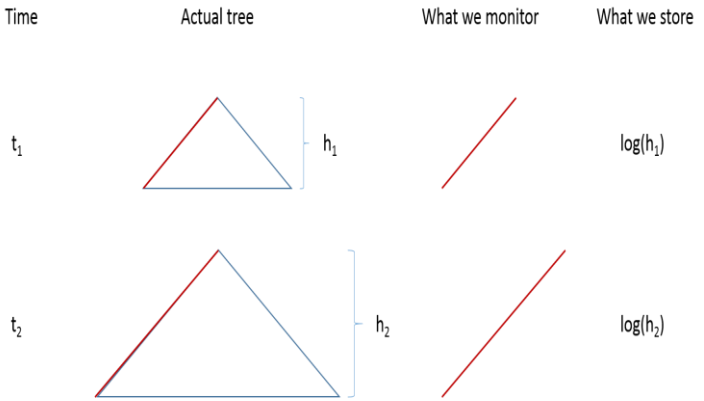
HYPERLOGLOG

17 (id) -> 2989035304 (hash code)
00010100111100001001010001001101 (hash code in binary)

2732191004
00111000101101111001101101000101

1704704513
10000000010111011101100110100110





HYPERLOGLOG BASIC IMPLEMENTATION

m = 2^p (for example p = 14)

counters = Array.ofDim[Double](m)

```
def add(obj: T): Unit ={
  hash_code = hash(obj) //use 64 bit hash code
  index = first p bits of hash_code
  w = last (64 – p) bits of hash_code = 00…01… (the number of leading “0” can be zero)
  sigma = 1 + number of leading zeros in w
  counters(index) = max(counter(index), sigma)
}
def getCount(): Double= {
        m*Array.average(counters.map(h=> 2^h)) //does not work well, see next slide
}
```

HYPERLOGLOG: IMPROVED ESTIMATION

```
def getCount(): Double = {
   correction = 0.7213/(1 + 1.079/m) //bias, computed empirically by google implementation
                                     //not by a formula like here
   correction * m * Array.harmonic_mean(counters.map(h=> 2^h))
}
```

For small numbers of distinct values, HyperLogLog over-estimates. Correction shrinks the estimate.

Try with and without correction on our test case.

LINEAR COUNTING

HyperLogLog does not work very well when the number of distinct elements is small.

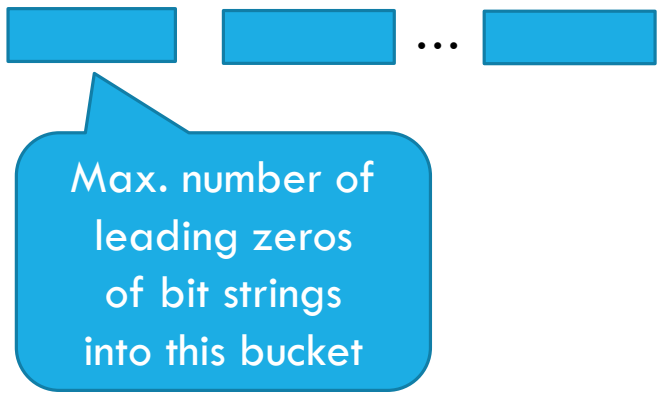
(absolute error vs. relative error)

Why? Given that we've seen k zeros at the beginning of a number, what's the chance of seeing (k + extra) at the beginning of a number? Does not depend on k, but if k is big, the fluctuation has very small effect because it's relative to k.

Solution: use a different algorithm (Linear Counting) for small number of distinct elements.

p = 14

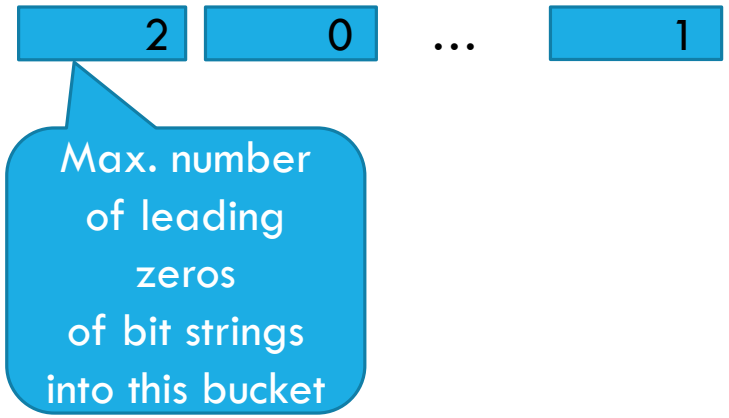
m = 2^p = 16384 counters



LINEAR COUNTING

p = 14

m = 2^p = 16384 counters



MAIN OBSERVATION

For small number of distinct elements, most buckets are empty. Nothing is hashed to them. So, the number of buckets that have 0 is more indicative of the number of distinct elements, than the actual values in the buckets.

Distinct elements = - m * log(zeros/m)

LINEAR COUNTING FORMULA DERIVATION

k distinct ids

m buckets

p = prob. a values goes to a particular bucket = 1/m because each bucket is equally likely

q = out of k distinct ids, no id goes to a bucket = (1 – p)^k

Z = 1 if bucket is empty and 0 otherwise

E[Z] = 1*q + 0*(1 – q) = q

Expected number of empty buckets = m * E[Z] = m * q = m (1 – 1/m)^k = m * exp(-k/m)

Solve for k: log(zeros/m) = - k/m, k = - m * log(zeros/m)

TASK 8: LINEAR COUNTING IN HYPERLOGLOG

```
def add(): Unit //as before

def getCount(): Double= {
        val zeros = counters.filter(_== 0).length //how many counters are zero
        - m log(m.toDouble/zeros)
}
```

What happens if zeros = 0 (i.e. we have no empty buckets)

IMPROVING THE MEMORY USAGE OF HLL

1. The data that HLL++ keeps is just the counters
2. Observation 1: For small distinct counts, most of the counters are zero. So we don't need to materialize them
3. Observation 2: Even when the counters are not zero, they store the height of the binary tree. Since we are considering 2^(64 – 14) max. distinct elements, this means the counters will store max (64 – 14). This number is represented in less than log(64 – 14) = 6 bits.

TASK 9: USE ELASTIC SEARCH AGGREGATION

ABOUT HASH FUNCTIONS

https://code.google.com/p/smhasher/

Each bit of the output has to have a prob. ½ of being 0 or 1

We have to see all 256 values of a byte roughly the same number of times

REFERENCES

- https://highlyscalable.wordpress.com/2012/05/01/probabilistic-structures-web-analytics-data-mining/
- Elastic Search Implementation https://github.com/elastic/elasticsearch/blob/6e3a4e21a118b73be2dfd5dfc2c787df5b950253/core/src/main/java/org/elasticsearch/search/aggregations/metrics/cardinality/HyperLogLogPlusPlus.java