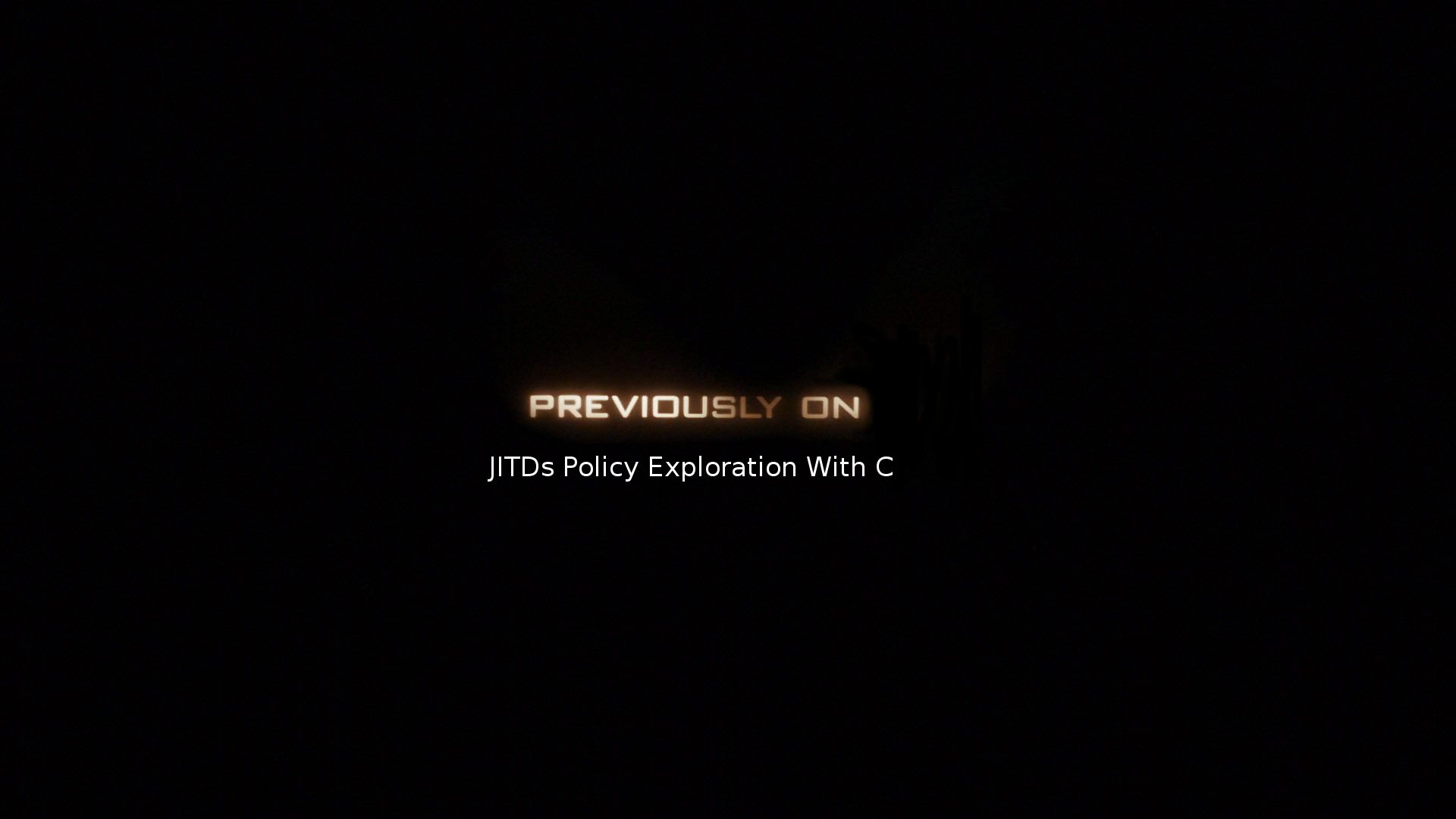
JITDs Policy Exploration C

Team Twinkle

Nov 4, 2015


PREVIOUSLY ON
JITDs Policy Exploration With C

And on today’s episode…

- How can we make things more interesting?
- Exploring a new paradigm of smart(er) JITDs + Policies
- Zipfian distributions & why they are interesting
- A proposed policy based on Zipfian and smart(er) JITDs
- Best case scenario and our actual aim
- A little bit of math & theory
- A bit more in depth look into the actual policy

Why switch things up?

Read… Splay… Read… Splay…

It works but it’s not very flexible.

… or interesting for that matter.

Let’s be a little clever!

- In fact, let’s make the JITDs clever!
- What if we could put metadata into cogs?
  - That could maybe work
  - But what kind of data could we put?
  - How can we use it in policies?

What kind of data should we encode?

- Can’t encode too much… bookkeeping is hard
- Should also consider that memory usage could get very large
- What data is actually useful and what isn’t?
- How can we encode data efficiently?

Think Reads!

- We have been working with read heavy workloads
- So how can we make decisions for them
- How about encoding read counts for BTree Cogs?
- Let’s do one better, let’s consider Zipfian Reads.

Zipfian Distributions

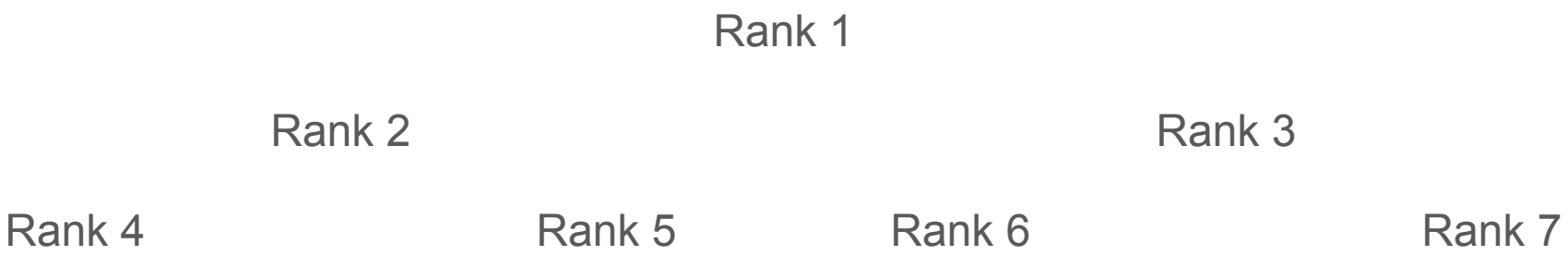
- Most data is naturally zipfian
- What does that mean?
  - certain elements in the data set dominate in terms of frequency
- For example - linuistics/web-serach/etc.
  - the most frequent word in english language is “the”
  - the most frequent word will occur approximately twice as often as the second most frequent word, three times as often as the third most frequent word, etc.
- Think in terms of ranks!

How can we take advantage of this?

- We have splaying now and we use it to balance the tree
- We are adding read counts:
  - we can now also put frequently read items at the top
- This should increase efficiency even further
- We can also be more flexible:
  - make smart(er) decisions about when to splay

The dream!

- If we can arrange the tree in a completely Zipfian manner that’d be great!
- What I mean is highly ranked (high frequency terms) will go at the top
- Similarly low frequency terms will go towards the bottom



Alas! ‘Tis but a dream...

- Items are ranked based on key, so we can’t only consider reads.
- We can’t quite reach the dream
- But we can get close

Let’s just put highly ranked items towards the top

- Think some cumulative distribution function (CDF) value defined by a DBA
- We don’t need to arrange it perfectly on average this should do pretty well
- We don’t need to worry about low probability items towards the bottom
  - They happen infrequently so they should simply create a bit of noise

Calculating number of elements based on CDF

- Given a predefined CDF (by a DBA):
  - we need to know what part of the tree to consider
- Let’s find the number of elements for the given CDF
- Ceiling of log_2 of that number will give us number of levels of tree

Let’s do some math… let’s define some stuff!

- Harmonic numbers: $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}.$
- Zipfian frequency:
  - for rank k
  - N data elements
  - s is a zipfian constant (assume 1 for sake of presentation)

  $f(k; s, N) = \frac{1/k^s}{\sum_{n=1}^{N}(1/n^s)}$ $f(k; s, N) = \frac{1}{k^s H_{N,s}}$
- Euler’s constant: $\gamma = \lim_{n\to\infty} (H_n - \ln(n + 1/2))$ $H_n \sim \gamma + \ln(n + 1/2)$
- n is the number of elements based on CDF we are looking for

Let’s derive the formula to find n

- Zipfian CDF: $\frac{H_{k,s}}{H_{N,s}}$
- CDF is defined by DBA so the only thing we don’t know is H_k
- H_k is thus = CDF * H_N
- From this and the formulas on last slide we come up with:
- n ≈ e ^ ((CDF * H_N) - gamma) - .5

Example:

- For a 100000 elements and CDF .5:
  - n = 236.4 levels = 8
- CDF .6
  - n = 793.3 levels = 10
- CDF .7
  - n = 2658.8 levels = 12
- CDF .8
  - n = 8908.8 levels = 14

Great!!! We don’t need to really look at a lot of data.

- We can now efficiently find what data to look at
- Now we just need to describe our policy
- We also need to describe bookkeeping:
  - How do you crack with read counts
  - How does one splay with read counts

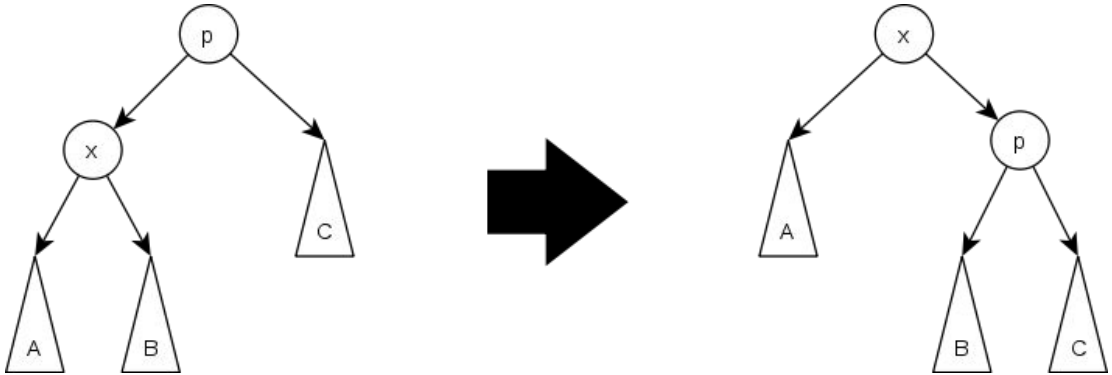
Read with cogs!

- We keep a cumulative count of reads at every cog for itself and its children
- This is clever encoding as we can determine some cool things from this:
  - Reads at a given cog (given cumulative reads - (left + right))
  - We can navigate around the tree to find highly read items (or clusters)

- For bookkeeping: Since cracking operations are recursive we need to just increment reads going down the tree in our recursion. This is efficient and elegant!

Splaying with cogs

- Splaying is done in zig, zigzag, and zigzig operations
- If we define read rearrangement rules for each we can recursively propagate the read rearrangement of the tree efficiently and elegantly like with cracking.
- For example for zig here reads of B, C, and A remain the same, reads for p are now = B + C reads
  Read for x = A + B + C



Putting it all together:

- Determine amount of levels to pay attention to (readjust if writes happen)
- In some self adjusting interval (can go very smart with this - think machine learning) run our rearrangement policy
- Move nodes up by looking at cumulative counts from top to bottom (subsets of trees)

  - This will give us both a pretty balanced tree as well as a somewhat zipfian distributed tree