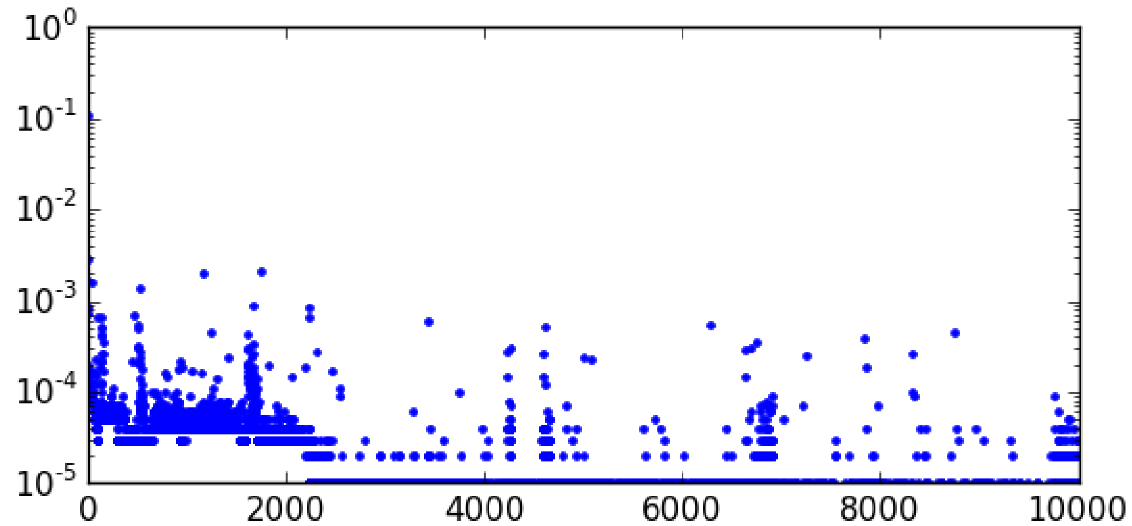
Policy Exploration for JITDs - Java

Team Datum

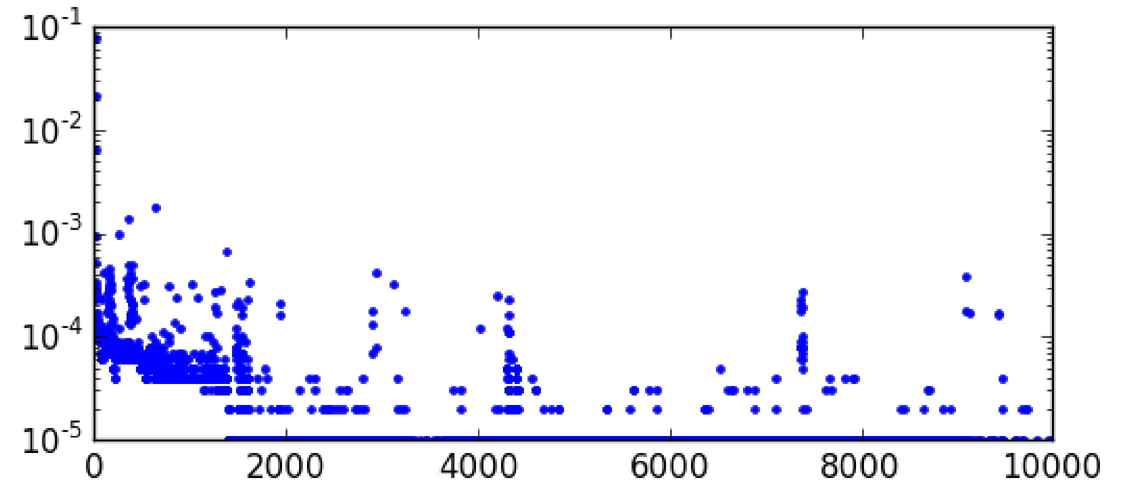
Splaying on Uniform Distribution

Cracking on Uniform Distribution
without splaying



Time Taken ~ 31.8ms

Cracking on Uniform Distribution
splaying for every 100reads

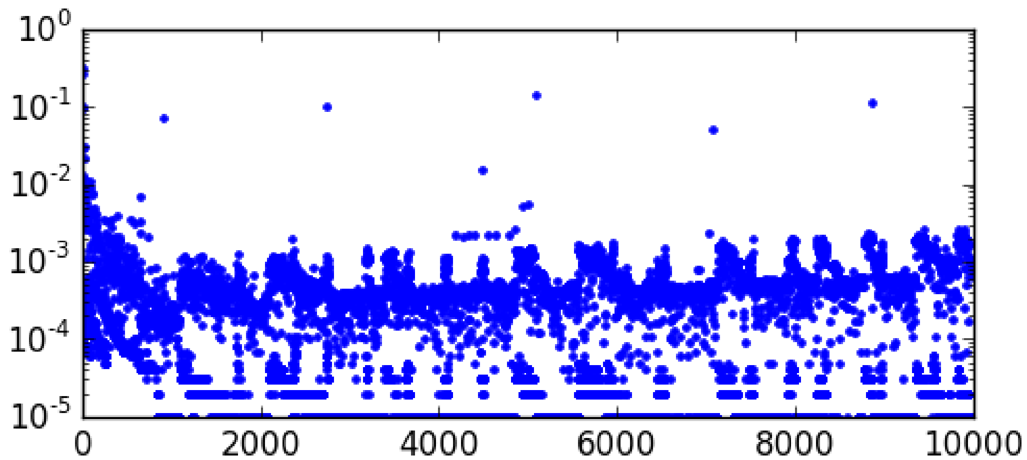


Time Taken ~ 26.8ms

Tested as
KeyRange 1000000
Load 1000000
Reads 1000

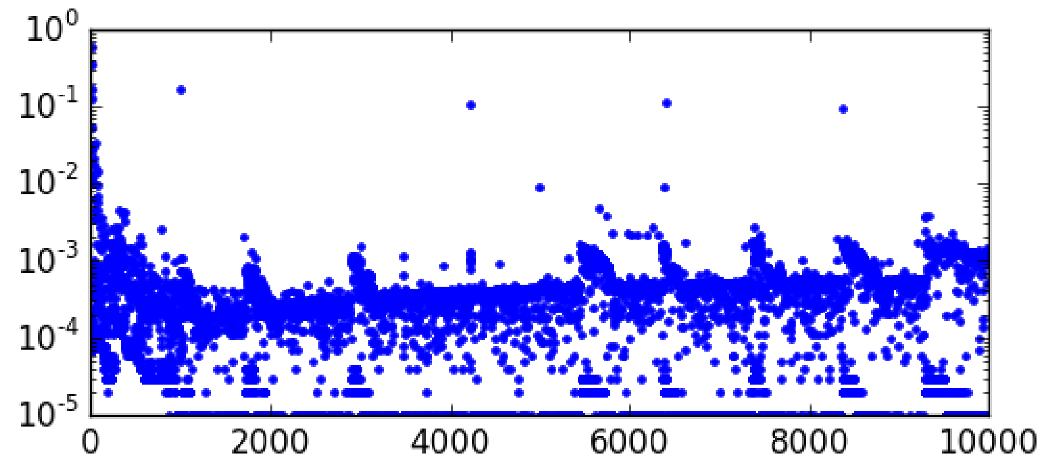
Splaying on Zipfian Distribution

Cracking on zipfian Distribution
without splaying



Time Taken ~ 421 ms

Cracking on zipfian Distribution
splaying for every 500 reads



Time Taken ~ 309 ms

Tested as
KeyRange 1000000
Load 1000000
Reads 1000

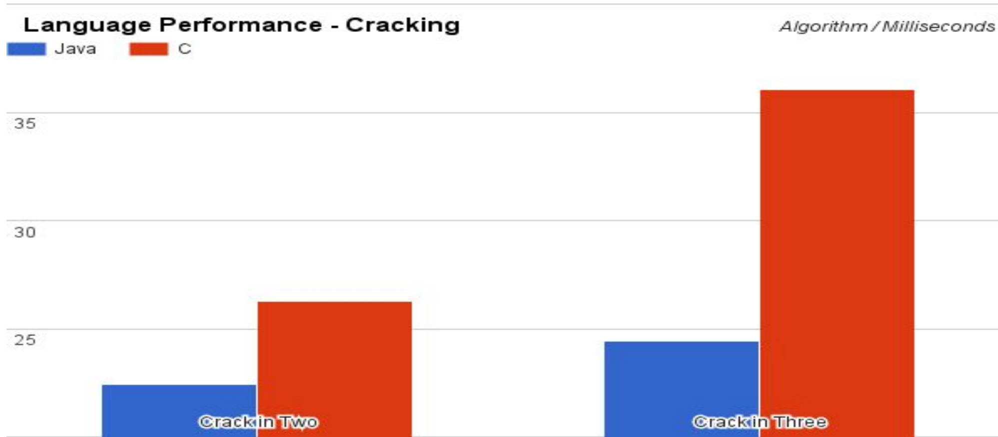
JITD C Group

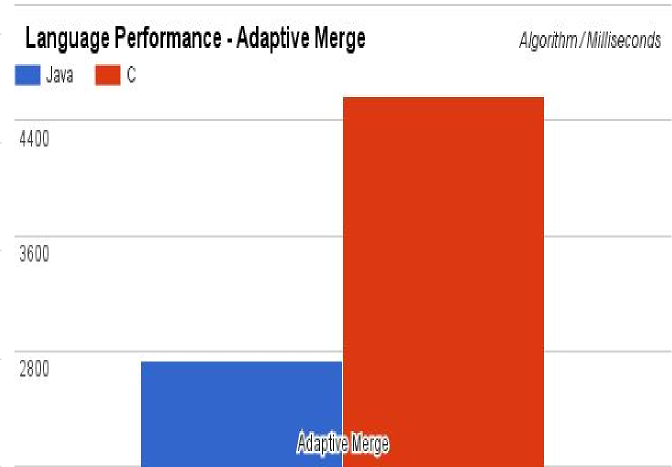
Alex, Razie, Aurijoy

Some Recaps

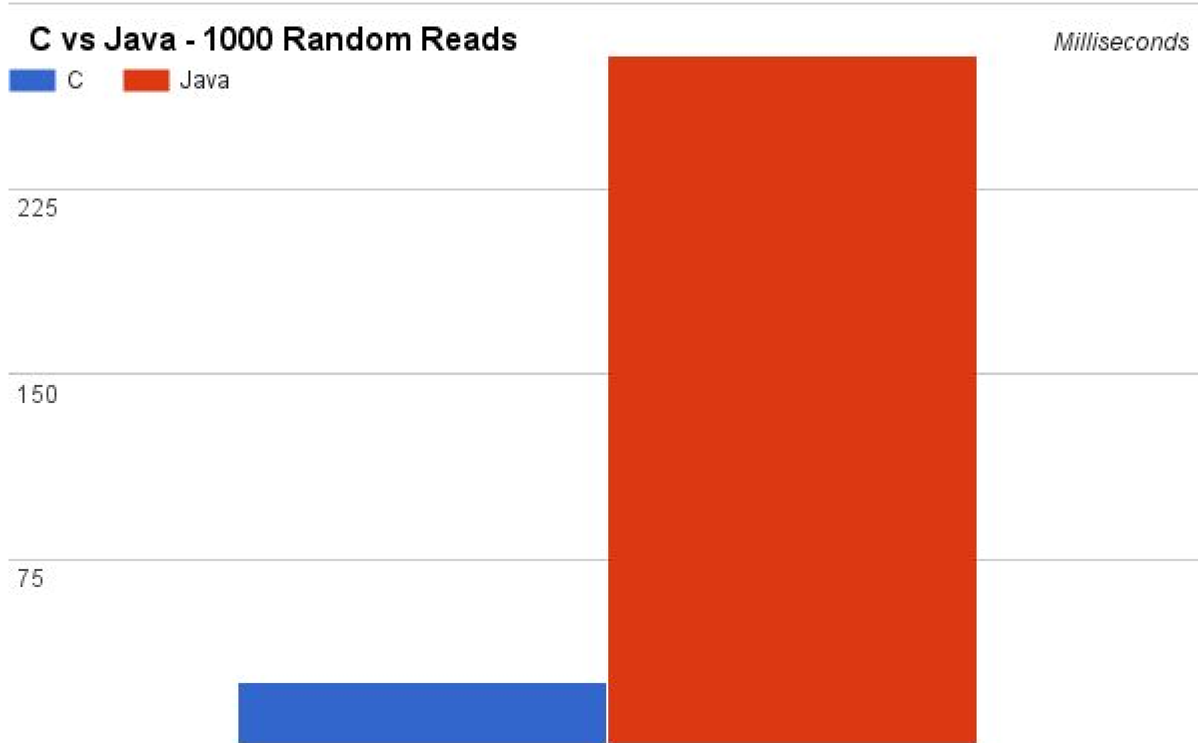
- Comparison between Java and C version for JITDs
- Splaying policy

Java and C Performances

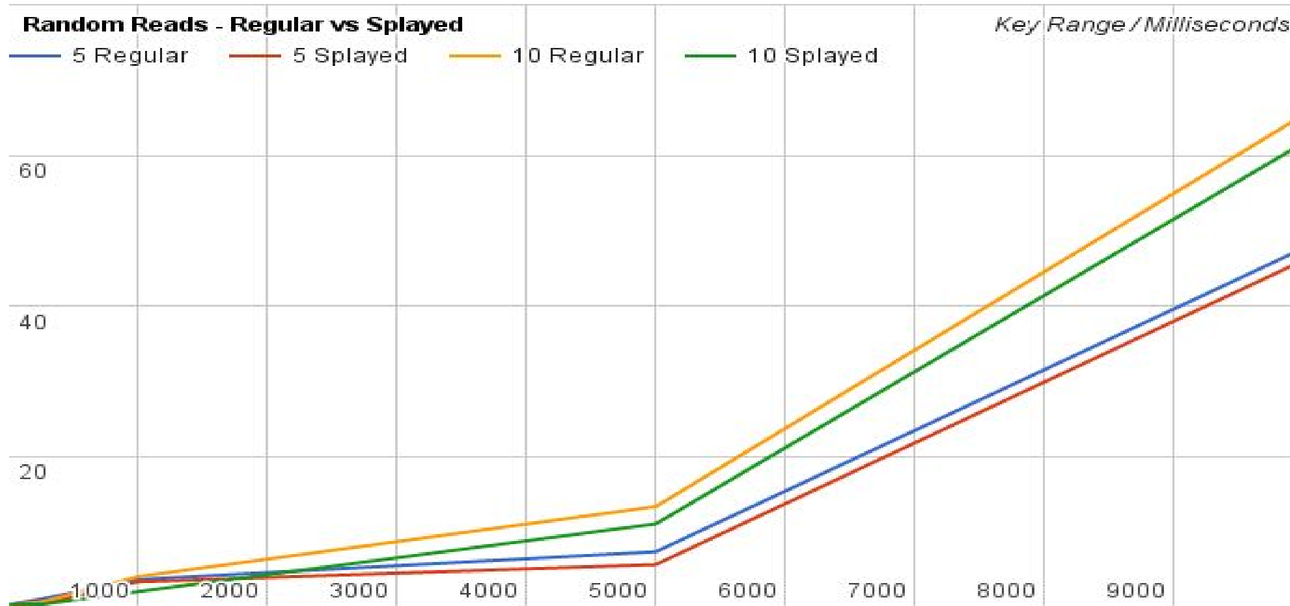




However over a 1000 Reads Things Are Better



Splaying policy- preliminary findings



Let’s see if Splaying by itself (at random) is good!

The setup:

- Buffer Size of 1000000
- Data is Randomly Distributed
- Key Range of 1000000
- Total Reads 10000
- We test splaying on every 250, 500, 1000 reads
- Our results here are the average of 5 separate runs (results were pretty consistent across runs)

How do we choose the random point to splay at?

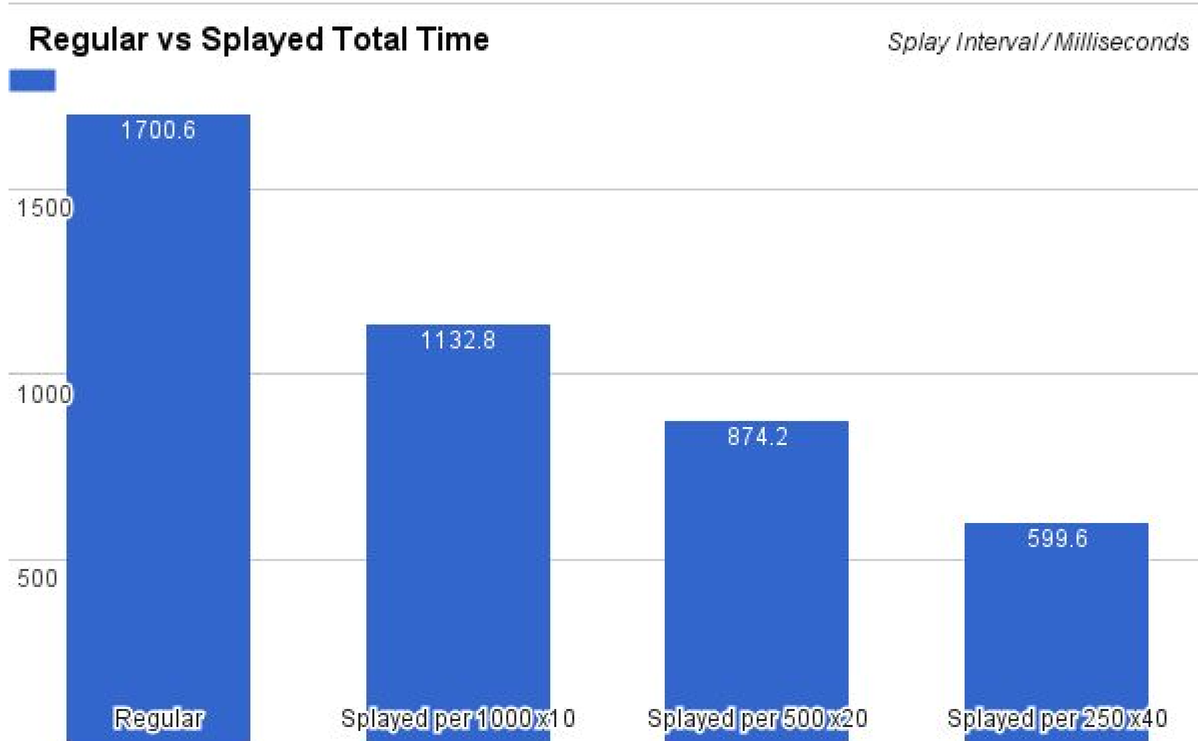
- We choose it while we do a read
- We single out the cog generated while cracking for the left hand side
- We use the cog generated for the read just before we splay

Why?

- It’s for free unlike finding the median
- It’s random

How does it perform per splay step?

How does it perform in terms of runtime?



Takeaway

- Splaying at random works great as splaying balances the tree pretty good regardless
- It may thus be better that our policy uses splaying as more of a balancing technique
- Splaying more often is better
- There is probably a cutoff point and we should find it

Tinkering with Splaying Interval- Variations of Splay Heuristic

Our efforts would be directed towards finding the variations over different splaying heuristics.

While the splaying policies used so far are not the ones used in canonical splay trees used widely.

We hope to get an idea of whether the intervals do matter over uniformly mapped zipf keys.

ReMapping the Keys in a Zipfian to have fair Prior over Tree Balancing

Previous week our choice of mapping the numbers generated by the zipfian had an initial bias. Keys with successive numbers had bias for being the actual successors in the balanced splay tree. So we decided to remove the bias by remapping the key-values after a shuffle. This would eliminate inconsistencies in the splay interval results over the zipfian.

Should we experiment with Dynamic Balancing Strategies

One of our major concerns in policy design is being able to guarantee bounded expectations over latency vs throughput.

So could we turn the problem over itself and by means of hierarchical balancing strategies to have guarantees on bounds. Our context so far has been read heavy workloads so our policies effectively translate into search structures.

Exploring More Interesting Workloads

While there is a tendency to design policies intended for different distributions remain high. We would like to point out that the most important distributions for our purpose are the ones naturally occurring as workloads.

So it is sufficient to say that our efforts are directed towards exploring important workloads and designing policies around them.

So far we have only modelled around a uniform and a zipfian distribution, we hope to find more important benchmark distributions from YCSB.

JITDs on Disk

Team Warp

Animesh, Archit, Rishabh, Rohit

Summary till checkpoint 1

- Explored and implemented different file formats
- Explored different ideas to store indexes on disk
    - LSM Trees
    - Paging

FILE FORMATS AND SAVING DATA TO FILE

- Different Cogs have different structure
- Using Visitors Pattern to write different Cogs
- Iterative algorithm to restore indexes/pages
- Two file formatters used
- Working on policies to use both the file formats in conjunction to avoid fragmentation

Detailed File Format For INDEX FILE As Stored in FILE SYSTEm

	DATA		SEPARATOR			DATA	
	COG TYPE	FILE NAME	ROOT FLAG	COG TYPE	VALUE	COG TYPE	FILE NAME
SIZE (BYTES)	2	50	1	2	8	2	50
TYPE	Char	Char[]	Bool	Char	Long	Char	Char[]

Cog Type	Meaning
A	Array Cog
B	BTree Cog
C	Concat Cog
E	Empty
F	File Cog
L	Leaf Cog
S	SubArray Cog

Detailed File Format For INDEX FILE As Stored in FILE SYSTem

	DATA		SEPARATOR		DATA	
	COG TYPE	FILE OFFSET	COG TYPE	VALUE	COG TYPE	FILE OFFSET
SIZE (BYTES)	2	4	2	8	2	4
TYPE	Char	Integer	Char	Long	Char	Integer

Cog Type	Meaning
A	Array Cog
B	BTree Cog
C	Concat Cog
E	Empty
F	File Cog
L	Leaf Cog
S	SubArray Cog

LSM Trees

- Timely flushing index tree in memory to disk
- Merging these files together to main index file
- Problems
  - Merging was very complicated
  - Restoring partial trees based on queries were problematic

PAGING

- Refined the concept of saving and restoring partial indexes into the concept of paging
- Page-In indexes based on queries
- Page-Out indexes based on available memory
- Current Progress
  - Bug Fixing
  - Coming up with benchmarks
  - Policies based on which pages should be paged-out

QUESTIONS?