Policy Exploration for JITDs (Java)

Team Datum

Testing Current Implementation
(On Zipfian Read Heavy Workload)

- The Current implementation has been tested against Distribution following Zipfian Workload with bulk number of Reads.

- The Experimentation is done against all three policies :
  - Cracking
  - Adaptive Merge
  - Swap (Hybrid policy)

- Parameters Used :

  Dataset size : 1 Million records

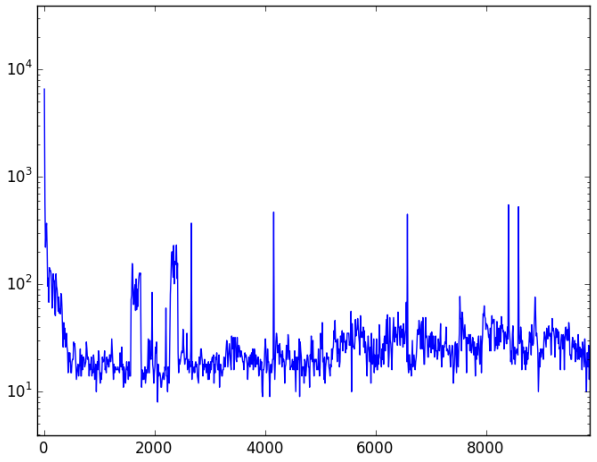
  KeyRange : 100000

  ReadWidth : 1000

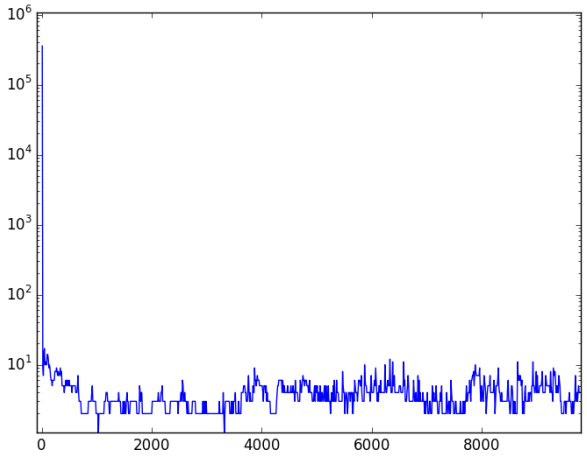
  Total Reads : 10,000 (Follows Zipfian distribution)

Cracking Vs Merging (on Zipfian)

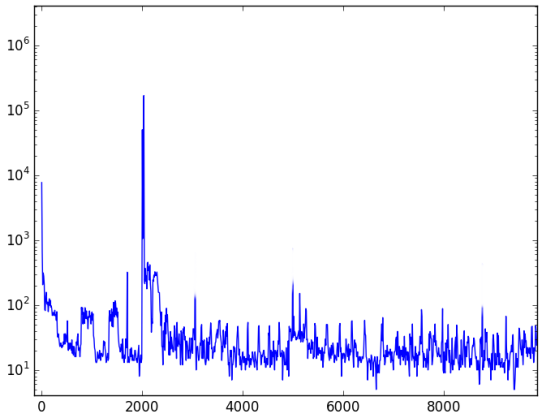
- Graphs have been plotted on Number of Reads / Avg Time Taken over 5 Sequential Reads (in MicroSecs)
- Cracking has low initial cost but takes more time to crack and answer the Range Query on the space that is not cracked before. (on low-probable range query reads)
- Whereas adaptive Merge will have more Upfrontcost but converges faster to even answer the low-probable queries quickly.
- Swap will get better tradeoff between low upfront cost and answering the low probable queries fastly.



Cracking (438.810 ms)



Merge (3613.754 ms)



Swap (2962.503 ms)

Motivation to policize for Zipfian Workloads

- Zipfian Workloads represent the Real-time workloads.
  - High Probable Queries are less.
  - Low Probable Queries are more. (Long tail of Zipfian Disribution).

- Even though Swap policy gets the tradeoff to answer low-probable queries quickly, amount of time it takes to sort partitions in Middle of the transition from Cracking to Merging is high.

- So, Our policy will be to answer high-probable queries quickly and also to answer low-probable queries in comparable time.
  - Answer High-Probable queries quickly → Keep those Query bounds on the top levels of Tree. (How?)
  - Answer Low-probable queries in decent time → Balance the tree regularly (When?)

- Finally the needed changes to the existing implementation are :
  1. **Balancing the Tree.**
  2. **Keep Most-Frequently or Recently Accessed nodes on top levels of the tree.**

Why Splaying?

- Balancing :
  - Near Balancing Property with all operations in O(log n) amortized time.
  - No extra memory required like in AVL!

- Recently Accessed Nodes on Top :
  - After each Splay Operation, the element that is splayed prior will remain in the next higher level.

- Can be easily integrated with Current Cog based structure without changing the internal cog structures.

When to splay?

- Option - 1 : Constant Intervals of Time.
  - Eg., Per every 500 reads!
- Option - 2 : After Each read?
- Option - 3 : Varying Intervals of Time.
  - Eg., Splay aggressively for initial reads and slow-down afterwards to reduce the time to splay.

  Problems :
  - How to find the function that gives varying intervals? Logarithmic?

Which element to splay?

- Exploiting the user Query Statistics. (Implements Property 2.)
  - Something that 'C' group is doing now!
    - Keeping ReadCount at each BTree Separator and splaying the useful elements retrieved from the stats.
  - Simply splay around lower Bound element or Upper Bound of Range Query.

- Balancing the tree? (implements Property 1.)
  - Splaying around Median element of the existing tree.

Implementation

(When To Splay : Element To Splay)

- We have completed implementation of two policies on top of Cracking Mode.
  - Constant Interval : Lower Bound of Range Query
  - Constant Interval : Median Element

- Constant Interval : Lower Bound of Range Query
  - Eg., Splay the current Tree Structure for every 'n' reads using the previous read's lower bound.

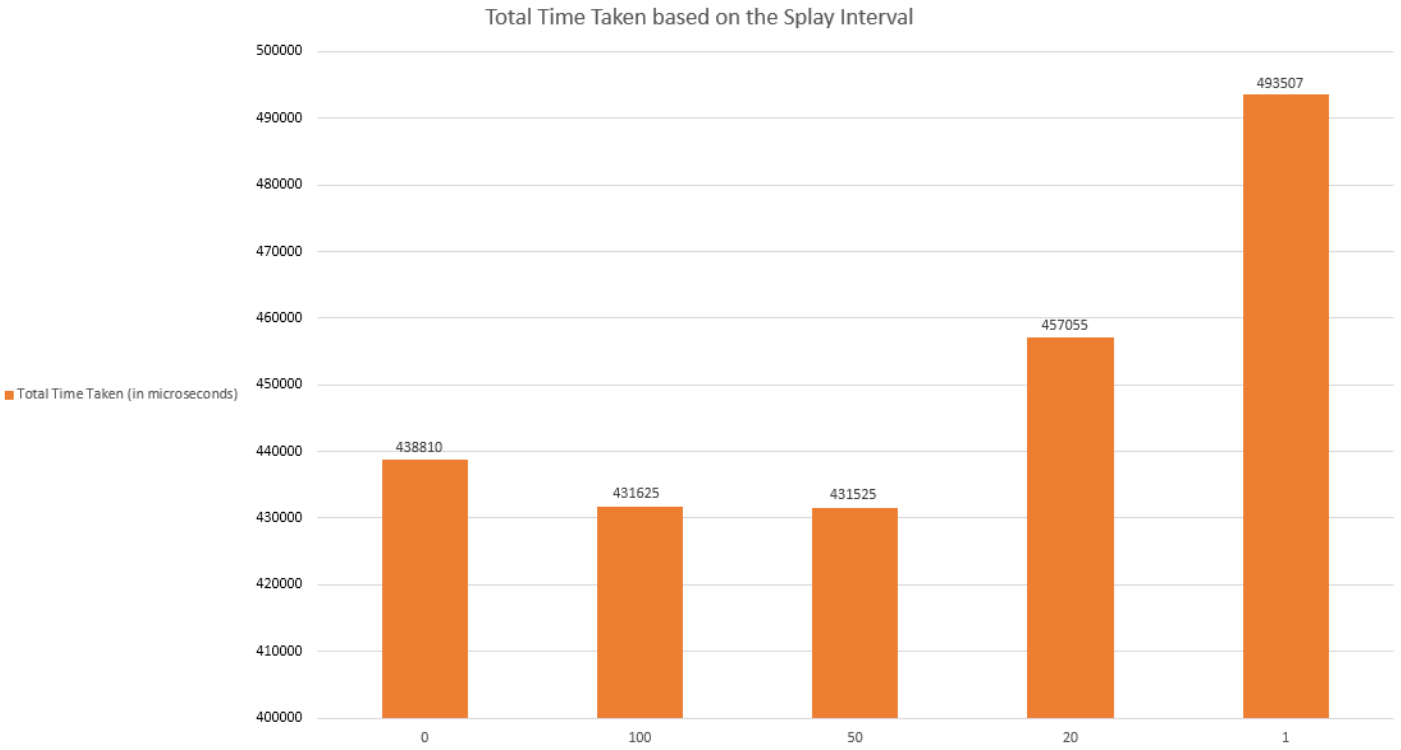
- Constant Interval : Median Element
  - Eg., Splay the current Tree Structure for every 'n' reads along the median element as root.

- All the experiments given are performed on :

  Dual core 2.40 GHz Intel i5
  8 GB of RAM
  Ubuntu 14.10 and JDK 1.7
  JVM heap size was set to 5 GB
  200MB of stack space.

Constant Interval : Lower Bound of Range Query

Each Y-value is the time taken for 10000 reads (in microsecs) and is averaged over 5 runs using data of size 1000000 following zipfian distribution in cracker mode on KeyRange 100000 and Read Width 1000.



Take Away of using Lower Bound to splay

- Performance improvement can be observed on splaying at 100 and 50 but performance began to deteriorate for splaying at 20 and per each read due to more time being spent on splaying.

- **Advantages** :
  - Simple to implement.
- **Limitations** :
  - There is no guarantee that splaying will actually improve the depth (may even increase the depth of the tree)
    - Eg., Suppose the 100th read is a low-probable one, if we splayed the tree on that element it can potentially skew the cog structure.

Constant Interval : Median Element

- Median Element is found in 2 Steps :
  - **Step-1** : Get Count('n') of BTree Separators in current structure.
  - **Step-2**: Do In-order traversal of Cog and return the element at 'n/2' position.

  Time taken for finding median : **(3/2).n**

- We can reduce the factor of time from '3n/2' to 'n/2' if we know the count. We can do this tweak if we can update 'n' during each BTreeCog Creation!

With vs Without Splaying - Changing Key Ranges

Each value is the time taken for 1000 reads (in microsecs) and is averaged over 5 runs on each type (with and without splay) using data of size 100000 following zipfian distribution in cracker mode with Read Width 1000.

Key Range	With Splaying (for every 10 Reads)	Without Splaying
10	5221	5809
100	8953	10585
1000	17332	18445

With vs Without Splaying - Changing Read Width

Each value is the time taken for 1000 reads(in microsecs) and is averaged over 5 runs on each type (with and without splay) using data of size 1000000 following zipfian distribution in cracker mode with Key Range 1000.

Read Width	With Splaying (for every 100 Reads)	Without Splaying
10	52715	53463
50	53441	55556
100	56537	58422
1000	58645	61177

Changing Splay Interval

Each value is the time taken for 1000 reads(in microsecs) and is averaged over 5 runs on each type (with and without splay) using data of size 1000000 following zipfian distribution in cracker mode with Read width 1000 and Key range 1000.

Splay Interval	Time Taken (in micro secs)
10	64980
50	61320
100	61041
200	64282

Take Away of using Median Element to splay

- Performance of splaying around median is better than splaying around lower bound of range query due to better balancing of tree.

- **Advantages** :
  - Tree is better balanced.
- **Limitations** :
  - Finding Median can be the bottleneck which consumes O(n) prior to actual splaying.
  - It is not making the best use of user's Read Statistics for answering high probable queries faster.

New policy!

1. Store the <Key, Value> pairs of <Separator, ReadCount> - For only the last R reads?

2. Get 'm' number of high frequent values in such a way that there will be at least 'e' difference with it's next highest value.

3. Perform splay on these 'm' separator values.

4. If ReadCount is increasing, we can re-scale the count values, by using a weightage factor 'w' (for example, w = 0.02) that multiplies with each value in the map and updates its ReadCount.

5. Repeat the above process and change the weightage factor 'w' variably if necessary.

Thank you!