2015

Policy Exploration for JITDs (Java)

FINAL PROJECT REPORT

TEAM DATUM

Srinivasa Reddy Bade sbade@buffalo.edu
Jayanth Nasika jayanthn@buffalo.edu
Madni Maksud Vadhariya madnimak@buffalo.edu
Venkata Sai Manoj Illendula villendu@buffalo.edu

Table of Contents

1. Overview

Just-In-Time-Data Structures (JITDs) is a new approach to switch between various Adaptive indexing schemes based on changing workloads. Briefly, underlying structure reacts differently to reads and write queries aiming to achieve best performance. JITDs use adaptive merge and cracking operations which act as two modes which can be switched one from another at any point without doing any special handling. This facilitates in faster retrieval and works better even over updates to the data.

With existing JITDs Java implementation given we have explored few policies targeting different kind of workloads. We have tested few policies using two read intensive workloads namely Uniform Distribution and Zipfian Distribution. We have integrated splaying operation which form basis of our policies differing in splay interval and element to splay on. We present experimental results which shows how splaying at right interval and on right element in JITDs structure improves performance gaining us early convergence than adaptive merge and with low initial cost as with cracking.

There are good amount of policies implemented on top of JITDs that can be explored. For the scope of the project we focus on policies incorporating splay functionality. We aim to splay on elements based on the number of times it is being accessed and hence further enhancement will be to splay on most frequently accessed/read elements at low cost.

2. How we began?

The Java Implementation for JITDs along with the paper was provided to us. Our initial goal was to mimic the performance plots to the one presented in the paper with similar experimental setup and parameters. This replication will not only help us better understand the existing implementation, but also makes us more aware of the areas where there can be a scope for improvement. To facilitate our testing, we developed a Python script which would plot graphs for the number of reads against the time taken for multiple runs.

We were successful and faced no issue in emulating the performance plot for the Cracking Mode but the plot for Adaptive Merge Mode was not matching the one as described in the paper. We tested the Adaptive Merge Mode by modifying the experimental parameters and found that it performs as expected when the data set was initialized with *1,000,000* records instead of *100,000,000* data records as presented in paper.

We tried to understand what led to the abnormal behavior when number of data records were *100,000,000* for Adaptive Merge Mode by profiling and confirmed that the JVM's Garbage Collector was not the cause for it. But instead, there was an attribute "*length*" which was computed very frequently in the BTree Cog and this was the major factor for the undesired behavior as it was utilizing more than *70%* of the total time for calculation. After a little research, we found that this could be avoided. After that, to emulate real world workload where most of time some of elements are accessed, we decided to use the Zipfian distribution in our testing for which we implemented the Zipfian KeyIterator. Implementation details will follow in later sections.

Testing current Implementation on Zipfian Read heavy Workloads:

The Current implementation has been tested against distribution following Zipfian Workload with bulk number of Reads. Goal is to understand the pain points in current implementation when Zipfian workload is applied. Experimented against all three policies (Cracking, Adaptive Merge and Swap policy) with dataset of size 1 Million records, keys distributed across *100,000* values on a total of *10,000* queries with read width of size *1000*. Graphs have been plotted on Number of Reads / Average Time Taken over 5 sequential runs (in microseconds).

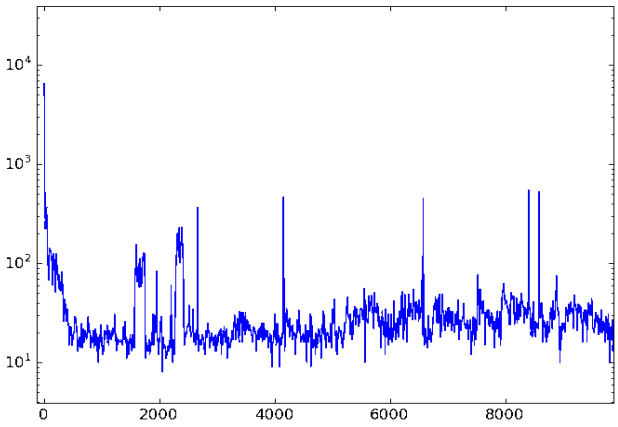


FIGURE 2.1: CRACKING

(X: no of queries, Y: time in microseconds)

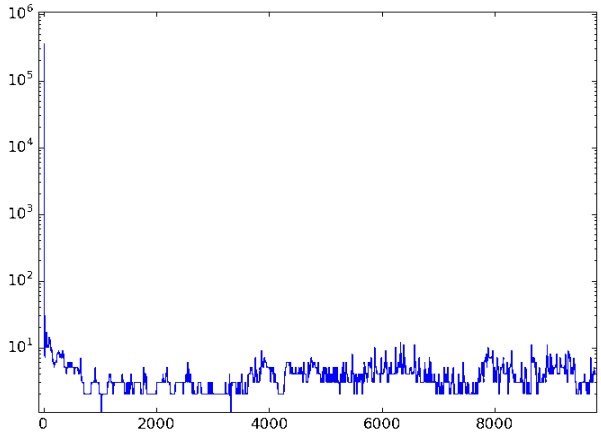


FIGURE 2.2: ADAPTIVE MERGE MODE

(X: no of queries, Y: time in microseconds)

As per the Figure 2.1 above, 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, as per Figure 2.2 will have more Upfront Cost to sort initial partitions but converges faster to even answer the low probable queries quickly.

As per the below Figure 2.3, Swap will get better tradeoff between low upfront cost and answering the low probable queries faster. Even though Swap policy gets the tradeoff to answer low probable queries quickly, the amount of time it takes to sort the partitions in between the transition from Cracking to Merging is high. So, cracking has been chosen as our underlying base policy that takes less time to answer any query on average.

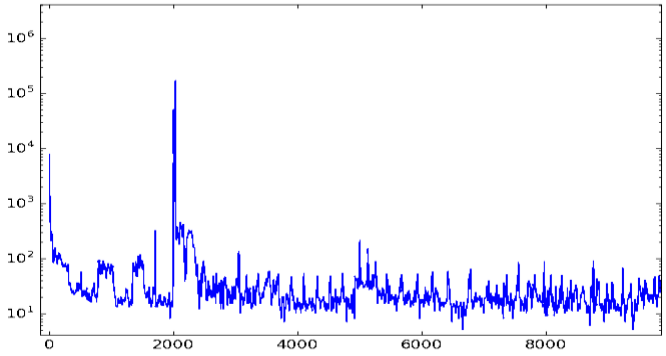


FIGURE 2.3: SWAP MODE

(X: no of queries, Y: time in microseconds)

Our policy is to answer high probable queries quickly and also to answer low probable queries in resonable time. **To answer high-probable queries quickly, it is required to place query bounds on the top levels of tree whereas to answer Low-probable queries in decent time, the tree structure needs to be balanced in regular manner.**

So our policy should be able to place most recently or frequently accessed elements at top and should balance the tree structure regularly. The next section outlines why splaying has been chosen to fulfill these properties.

3. Splaying: Its policies

3.1. What is splaying?

A splay tree is an efficient implementation of a balanced binary search tree that takes advantage of locality in the keys used in incoming lookup requests. Whenever an element is looked up in the tree, the splaying operation will reorganize the tree to move that element to the root of the tree, without breaking the binary search tree invariant.

There are three kinds of tree rotations that are used to move elements upward in the tree. These rotations have two important effects: they move the node being splayed upward in the tree, and they also shorten the path to any nodes along the path to the splayed node. This latter effect means that splaying operations tend to make the tree more balanced.

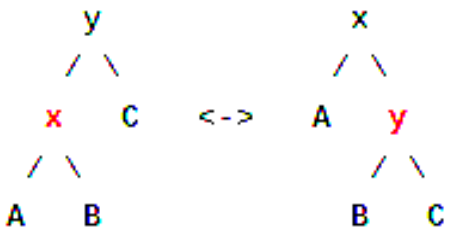


FIGURE 3.1: SIMPLE ROTATION

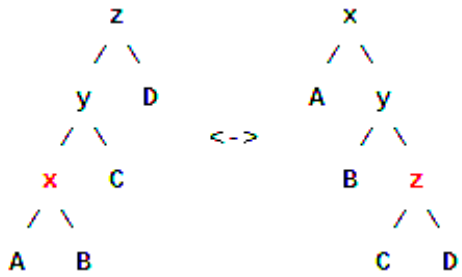


FIGURE 3.2: ZIG-ZIG OR ZAG-ZAG ROTATION

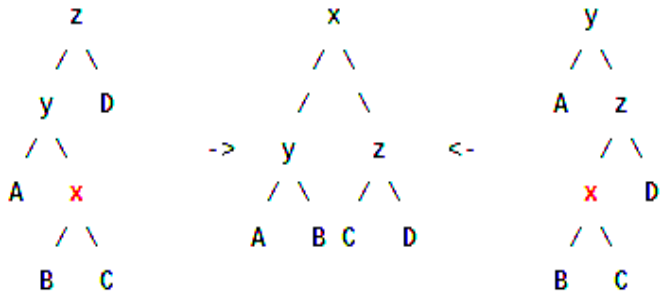


FIGURE 3.3: ZIG-ZAG ROTATION

Simple Rotation:

As shown in Figure 3.1, Simple Tree Rotation moves the splayed node x up to become the new tree root. Here we have *A* < *x* < *B* < *y* < *C*, and the splayed node is either x or y depending on which direction the rotation is.

Zig-Zig or Zag-Zag Rotation:

As shown in Figure 3.2, here tree rotations are performed in pairs so that nodes on the path from the splayed node to the root move closer to the root on average. In the "zig-zig" case, the splayed node is the left child of a left child or the right child of a right child ("zag-zag").

Zig-Zag Rotation:

As shown in Figure 3.3, in the "zig-zag" case, the splayed node is the left child of a right child or vice-versa. The rotations produce a subtree whose height is less than that of the original tree. Thus, this rotation improves the balance of the tree.

3.2. Why balance using splay and no other data structures?

As already specified in Section 2, in order to choose the data structure, it is necessary that it should be able to satisfy the above said properties like being able to balance the tree and should have mechanism that can keep recently accessed or most frequently accessed elements at higher levels of tree.

Some data structures that can balance the binary tree are AVL-Trees, Red-Black Trees, Splay Trees, Treaps. Out of these, AVL-Tree, Red-Black Tree are the best to balance with $O(\log n)$ average and worst case time. But, both the AVL and RedBlack trees require extra metadata to the existing TreeNode eg., the balancing factor for AVL and color of the element for RedBlack-tree.

Our goal here is, not to complicate the things, but to use the existing implementation optimally in the given period of time. Splay Tree best fits in this case where it also takes only $O(\log n)$ amortized time. Splay operation doesn't require any metadata and there is even no need of disturbing the current Cog based structure of the B-Tree node. As given in the previous section, after each splay operation, the element that is splayed prior will remain in the next higher level. This property ensures that recently accessed elements will always remain on the higher levels of the tree.

3.3. When to splay?

In order to use the existing implementation optimally, we have decided to implement the concept of splaying as an integral functionality rather than as a separate Cog. So, the first question to be addressed is when the splay operation should be performed, i.e. after how many reads or at what time interval, splay should be performed.

The question is not simple and straightforward, so we considered the following three options:

1. **Splay after each read**: Splaying after each read may seem to be an overkill, since more amount of time will be spent just to balance the tree rather than answering queries.
2. **Splay after constant intervals of time**: It would be better to splay at constant intervals of time, for example to splay every *100* or *200* reads. Challenge will be in choosing the best interval to splay that will be able to better balance the tree at minimal cost of splaying.

3. **Splay after variable intervals of time**: Even though splaying at constant intervals of time seems fine, one observation we encountered is that the amount of time it takes to splay increases over time. It is due to more nodes being cracked for each new query which in turn increases number of nodes in the tree. As the number of nodes ('*n*') increases, '$\log n$' will be increased over many reads. So, it may seem ideal to slow down the rate of splay as we progress. For example, splaying the tree aggressively like per every *50* reads until initial *10,000* reads and then increase the splaying interval by a factor '*k*' for the next *10,000* reads and so on.. This will reduce the time spent for splaying on heavy data sets.

Currently system supports splay for each read and for constant intervals of reads. Splaying after variable intervals of time will be experimented later.

3.4. What element to splay on?

The next question to be addressed is what element should be chosen to splay. The choice of the element to be chosen to splay depends on the properties that we want to exploit. For example, to get a better balancing of tree, it may seem wise to splay around the median element. Another option is to splay on the most frequently accessed elements. If we want to place recently accessed or most frequently accessed elements at the top, the only way we get to know about these statistics are by exploiting the previous queries posed by users.

The current implementation supports splaying around median and splaying around lower bound of the previous range query. We have been exploring some policies which can find the most frequently accessed elements in an effective way. These elements can be splayed to the top and as a result, they will answer high-probable queries effectively by balancing the tree.

A naive approach for splaying around most frequent elements is as follows:

- Maintain the read count values for each separator and increment it whenever it was accessed.
- After predefined interval of reads are executed, the splaying operation would splay on top 'n' most frequently accessed values.
- After splaying is done, the read count values can be reset. But will lose the information. Otherwise, we can have the values retained by modifying the readcount values of all the separators by reducing it with a factor of small value like *0.1* or *0.001* when any of separator readcount value is exceeding its maximum integer range.

Although this approach seems straightforward to implement using a Hashtable, but will be memory inefficient. In real time workloads following Zipfian distribution, most of the elements will have less number of occurrences (frequency), So, it appears wise to store and maintain only the most frequent elements rather than unnecessarily storing all the elements (i.e. the access counts of separators) in the cache.

The problem of finding most frequently accessed query bounds can be correlated with the problem of finding frequent elements over data streams. Many algorithms have been proposed for finding the frequent elements. The algorithm that we chose should fit our need. Some of the popular algorithms that are worth to experiment are:

1. **Lossy Counting**: This algorithm maintains tuples in memory, where each tuple constitutes of item, a lower bound on its count, and a "delta" (Δ) value which records the difference between the upper bound and the lower bound. At each read operation, it checks whether tuple corresponding to the item currently read exists. If it does, then it increments the lower bound count by 1, else it creates a new tuple for that item with lower bound count set to 1 and Δ set to n/Φ, where 'n' is total number of reads up to that moment and 'Φ' is user-defined ratio defining the minimum count of items to classify it as a frequent element.

For example, if we classify items as frequent only if the individual item read count is 20% of total reads, then Φ=0.2 . Periodically, all tuples whose upper bound is less than Δ value gets deleted on the bucket boundaries. Accuracy of this algorithm improves with the bucket size.

2. **Space Saving Algorithm:** This algorithm stores the most frequently accessed items in form of composite elements consisting of item and its count in a bounded size set. This set maintains pointer to minimum count element at any point in the set. With every read it checks if item is present in the set, if it exists its count is incremented by one else element with minimum count is replaced and this element is pointed as minimum.

3. **Count Min Sketch:** This algorithm maintains a sublinear space data structure which a matrix of size d x w, where w and d are defined in terms of error rate and probability of not exceeding error. Initially all the cells in the matrix are assigned with 0s. There are d pairwise-independent hash functions used which map to column values 1 to w with minimal collisions each belonging one row. With each read, each of d hash function generates a column number based on the item read and increments exactly one cell in each row by one. Total count of an item can be found by getting the minimum value of all the cells mapped by d hash functions. To get list of items which are frequent, heap can be maintained where items whose count is more than threshold value or Φ which will in terms of total read of individual item and total reads can be set by user. Here accuracy of results increase with value of d and w.

The parameters to be considered to choose the Best-fit algorithm that suits our needs will be provided in further sections.

3.5. Proposed Policies

Each policy is nothing but a combination of [“what to splay”: “when to splay”]. As said before, currently the following policies have been implemented:

1. Constant Interval: Splaying around Median
2. Constant Interval: Splaying around lower bound of range query
3. Constant Interval: Splaying only the most frequent elements.

4. Implementation

4.1. Zipfian Workload Generation

The Zipfian Distribution follows Zipf's law where frequency of an element in the distribution is inversely proportional to its rank. As per the empirical studies, in real world scenario, users generally query 10% of data (Highly-probable), 90% of time and vice versa. So, it is required to generate the real-world distribution like Zipfian distribution with being able to vary the skewness factor. Skewness defines the steepness of the Frequency-Rank curve. General Rule of thumb is, more skew factor implies less more-frequent items and long tail of less-frequent items.

The main goal is, Zipfian generator should have high throughput for generating more number of elements per millisecond and should have low space foot-print. The following are the options that are considered before finalizing the Zipfian generator.

4.1.1. Using trial and error method

$$f(k;s,N) = \frac{1/k^s}{\sum_{n=1}^{N}(1/n^s)} = \frac{1}{k^s H_{N,s}}$$

This method will first pre-calculate $H_{N,s}$, the N^{th} generalized Harmonic Number. It is nothing but summation of inverses of given range of elements (*n*) to the power of skewness factor (*s*).

Each time a new element is to be generated, random element rank *k* is generated and its frequency is calculated using the above formula, Frequency ($f(k;s,N)$) indicates the fraction of the time the k^{th} most common word occurs in current distribution. Another random double value (*dice*) is generated and compared against the calculated frequency value. If the calculated frequency value *f* is greater than the random generated *dice* value, return the corresponding rank *k*. Else, repeat the process until *f* > *dice* value.

This whole process will ensure that the values being generated follows Zipfian distribution. The disadvantage of this method is it is naive and takes large amount of time to calculate as it sits in tight loop to return the associated element (rank).

4.1.2. Using Hashmap to store Zipfian values

In this method, Navigable Hashmap is used to store all the possible associated Frequency values vs Rank (Element) instead of iterating through loops several times to find the rank. The map values are pre-calculated and whenever a new element is to be generated, it just finds random double element (*dice*) and queries the map to return associated element or rank.

4.1.3. Using YCSB's Zipfian generator

There is an existing Zipfian Generator in Yahoo! Cloud Serving Benchmark (YCSB) open-source program suite. It uses the algorithm given in the paper "Quickly Generating Billion-Record Synthetic Databases" By Jim Gray et al. The program presented here to generate a Zipf distribution uses constants alpha, zetan and eta derived from theta (skew factor) and *n* (Distribution size). The function zeta returns the Harmonic Sum ($H_{N,s}$) discussed previously.

4.1.4. Experimental Results: Which is preferred?

We have tested all the above mentioned three methods over *100* keys and Skewness as *0.99*. Their Key Distribution is plotted as Key (Rank) vs Number of Occurrences (Frequency). It is evident from the graph that all the methods have generated the Zipfian Values as expected with inverse relation between Rank and their frequency.

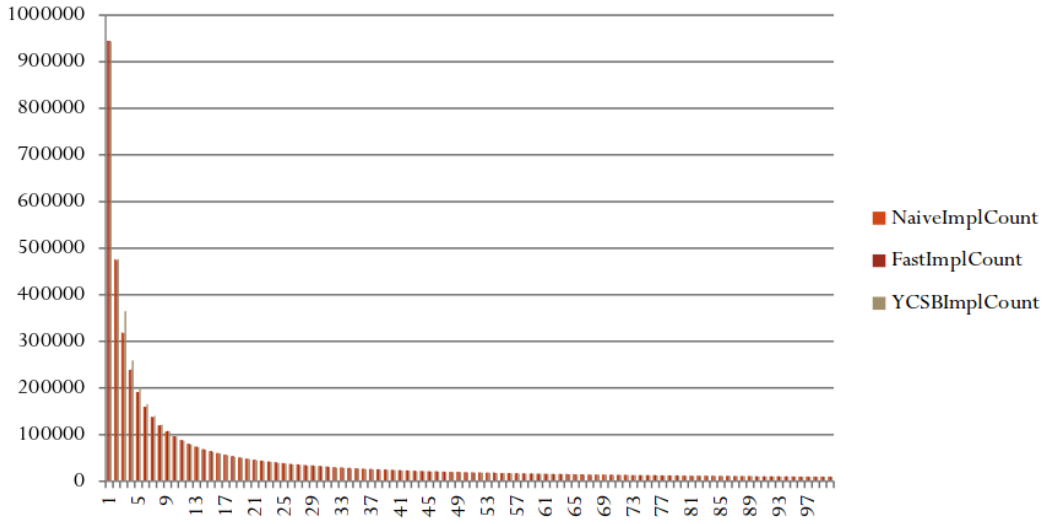


FIGURE 4.1: RANK VS. FREQUENCY

Even though all the methods are accurate in generating the Zipfian distribution values, they vary by the time and the amount of space they use in the process of generation. The following is the average amount of time taken by 3 methods in generating 2 million Zipfian values:

Implementation	Method	Time Taken (in millisecs)
Naive	Uses Trial and Error Method	67212.71
Fast	Uses Hashmap to store Zipfian values	540.02
YCSB	Uses YCSB's Generator	950.11

It is evident from the values that the method using HashMap to store Zipfian Values is outperforming others as it is storing values prior. The possible downside is, it takes much more space to maintain all the values in Map compared to other methods, which almost takes negligible space. It is not wise to consider the first method since, it is nearly 12 thousand times slower than other two methods.

So, taking both space and time taken into account, YCSB Implementation has been chosen. One more important use is that Scrambled YCSB Zipfian generator can generate numbers that are not in sequence but follows zipf's law, by hashing the generated values over predefined range.

4.2. Splaying Implementation

This section gives implementation of splaying on the current Cog based Structure which is introduced in Sec-2.1. As said before, splaying moves the required BTree node to the top of the tree as root, by following a sequence of operations like either zig-zig or zig-zag and finally it will perform the zig or zag operation based on the alignment of node w.r.t. the current root node.

We have implemented the SPLAY algorithm in bottom-up fashion, that is, first we navigate to the node that we need to splay on, if it exists, we will move that node up through a series of Rotate Right or Rotate Left operations.

The following are the mappings between traditional splay tree and our Cog-based Splay tree:

Traditional Splay Tree Notation	**Current Implementation Notation**
Binary Tree Node	BTree Cog Node (cog)
Node Key	BTree Cog separator value (separator)
Zig Move	Rotate Right(RR) or Rotate Left(RL)
Zig-Zig Move	Two sequential Rotate-Right (RR+RR) (or) Two sequential Rotate-Left (RL + RL) operations.
Zig-Zag Move	Sequence of Rotate-Right operation and Rotate-Left Operation (RR + RL) (or) Vice versa (RL + RR) operations

Splay Algorithm:

Splay Algorithm takes the current Cog structure (Tree) and Key to be splayed (Moved s new root) as input. Algorithm essentially follows the 2-Step process:

Step - 1: Traverse to the target BTree Cog, using input Key value.

Operation: It operates by traversing through the nodes starting from root until it finds the target splay node (BTree Cog). The process is similar to the one used for searching a node in Binary Search Tree (BST) i.e., It compares the separator value of current BTree Cog against input key and if *cog.separator* is greater than input key, it moves to the left sub-tree *(cog.left*) by calling *SPLAY(key, cog.left)* else it moves to the right sub-tree. This process is recursive and is repeated until it finds the target BTree Cog.

Algorithm 1 Splaying algorithm

```
1: procedure SPLAY(key, cog)
2:     if cog is BTree then
3:         if cog.separator > key then
4:             if cog.left is BTree then
5:                 if cog.left.separator > key then
6:                     cog.left.left ← Splay(key, cog.left.left)
7:                     cog ← RotateRight(cog)
8:                 else if cog.left.separator < key then
9:                     cog.left.right ← Splay(key, cog.left.right)
10:                    cog.left ← RotateLeft(cog.left)
11:                else
12:                    cog ← RotateRight(cog)
13:                end if
14:            end if
15:        else if cog.separator < key then
16:            if cog.right is BTree then
17:                if cog.right.separator > key then
18:                    cog.right.left ← Splay(key, cog.right.left)
19:                    cog.right ← RotateRight(cog.right)
20:                else if cog.right.separator < key then
21:                    cog.right.right ← Splay(key, cog.right.right)
22:                    cog ← RotateLeft(cog)
23:                else
24:                    cog ← RotateLeft(cog)
25:                end if
26:            end if
27:        else
28:            return cog
29:        end if
30:    end if
31: end procedure
```

Step - 2: Apply sequence of Rotate operations to splay node back to root.

Operation: The decision of which rotation to be performed depends on the alignment of the child BTree cog w.r.t its parent cog. Suppose the current cog is the left child of its parent cog, current cog should be rotated right by calling *RotateRight(cog)* to move up by one level. This operation is same as the traditional Zig operation.

Similarly, if the current cog is the right child of its parent cog, current cog should be rotated left by calling *RotateLeft(cog)*. This sequence of rotate operations are repeated until the target BTree cog is splayed as the root of the input cog.

Algorithm RotateRight algorithm

```
1: procedure ROTATERIGHT(cog)
2:     if cog is BTree then
3:         tempcog ← cog.left
4:         cog.left ← tempcog.right
5:         tempcog.right ← cog
6:         return tempcog
7:     end if
8: end procedure
```

Algorithm RotateLeft algorithm

```
1: procedure ROTATELEFT(cog)
2:     if cog is BTree then
3:         tempcog ← cog.right
4:         cog.right ← tempcog.left
5:         tempcog.left ← cog
6:         return tempcog
7:     end if
8: end procedure
```

4.3. Implementation of Proposed Policies

4.3.1. Constant Interval: Lower bound of range implementation

The current policy should splay the cog at constant intervals time using Lower bound of range query as the key. The implementation of the policy is simple since a simple mathematical modulo operation is enough to check whether to splay or not after each read. For splaying, it is only needed to pass the lower bound provided by the user as key and the *driver.root* as cog. Experimental results that compare before and after splaying affects, follows in the next section. Limitation of this policy will be, there is no guarantee that splaying will actually improve the depth (Balance the tree).

Example: Consider the keys of '*val*' column in Table '*A*' Range from (1 - 10,000) and splaying is performed for 100 reads. i.e., Suppose 100th query(read) is *SELECT * FROM A WHERE A.val >=990* ; and the element to splay is 990 (at extreme end) and let that even be low-probable query then, if the tree is splayed on key 990, there are two disadvantages :

1. There is no use in splaying as it will not balance the tree.
2. As it is not even a High-probable query, there is no point in keeping the cog at top-levels of tree.

From the example, it is clear that even though this policy is simple, in worst case, there is chance that performance will deteriorate due to the time being spent on splaying without actually doing any useful work (i.e., either balancing or keeping the most accessed elements at top-levels of tree)

4.3.2. Constant Interval: Finding Median

Exploiting the advantages of balancing a tree, splaying at the Median element seems to be the better alternative to splaying at lower bound. Implementation to find Median element takes 2 steps:

Step-1: Get Count(n) of BTree Separators in current Cog structure.

Step-2: Do In-order traversal of Cog and return the element at $\frac{n}{2}$ position.

Time taken for finding median: ${}^{3n}/_{2} = O(n)$

```
public static Long findMedianBTreeSeperator(Cog cog, int n) {
    if(cog != null && cog instanceof BTreeCog) {
        Long med = findMedianBTreeSeperator(((BTreeCog)cog).lhs, n);
        if (med != null) return med;
        if (inorderNodeIndex == n/2) return ((BTreeCog)cog).sep;
        inorderNodeIndex++;
        return findMedianBTreeSeperator(((BTreeCog)cog).rhs, n);
    }
    return null;
`}
```

Even though this approach seems fine, we can reduce the time complexity from the current $^{3n}/_{2}$ to $^{n}/_{2}$ if we already knew the count. This tweak has been implemented by including a static variable 'separatorCount' in Crack Mode where the variable is incremented every time a new BTree Cog is created. Thus we can avoid traversing the tree every time to find count of BTree Nodes. But, this may not hold true, if there are any deletes or updates to the tree.

The following is the transformation of the Cog structure before and after splaying around the median element:

4.3.3. Constant Interval: Most Frequently Accessed (MFA) Elements

From the survey paper [5] it has been established that it is optimal to find Most Frequently Accessed elements using either Lossy Counting, Space Saving or Count Min-Sketch algorithm considering the following parameters in their order of preference:

1. Precision and Recall (Nearly 100% is expected).
2. Amount of time it takes to find frequently accessed elements. (should be comparatively less w.r.t actual query execution time)
3. Size of traces that needs to be maintained to find frequently accessed elements. (Should be comparatively less w.r.t space needed for storing the Cog structure.)

The following are the steps that are performed before actually testing the performance benefit of splaying around most frequently accessed elements:

Step-1: Computation of precision and recall for each of the implementation (LC, SSL & CMS) against different Zipfian Read Heavy Workloads.

Step-2: Computation of overall execution time for each implementation to return the MFA elements.

Step-3: Estimation of space required for these implementations to return the MFA elements.

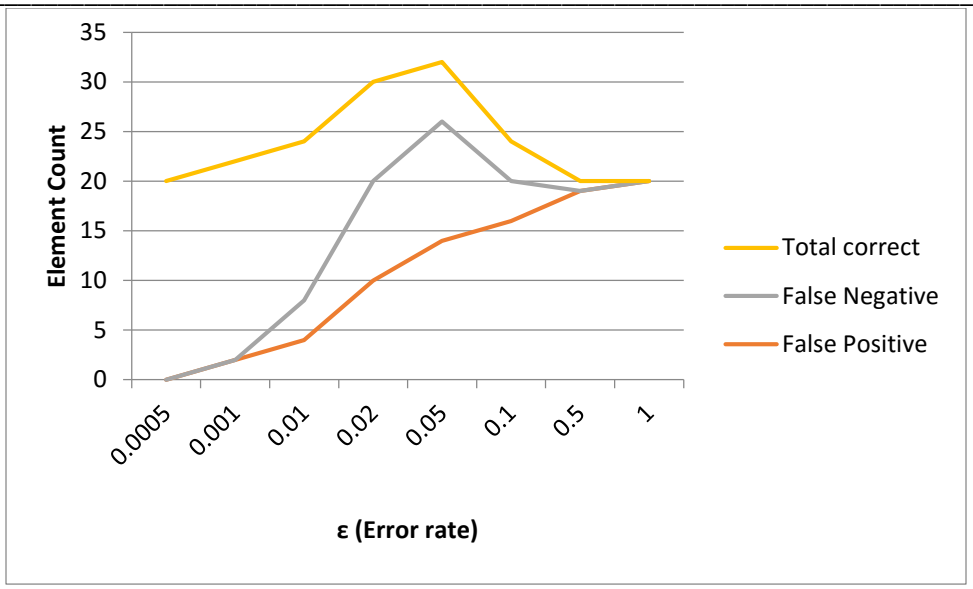
We have used open-source library named Streaminer[6] that provides implementations for algorithms like Lossy Counting (LC), Space Saving using Lists (SSL), Count-Min Sketch (CMS) etc. The below steps show the detailed description of the approach followed to experiment the above mentioned steps 1-3:

Step-1: Computation of precision and recall for each of the implementation (LC, SSL & CMS) against different Zipfian Read Workloads.

We tested the precision and recall evaluation on implementations of LC and SSL against the following Workload:

Scrambled Zipfian Distribution (Skew factor: 0.99)
Reads: 10,000
Key Range: 1000
Data Size: 1000000
Each Method (LC, SSL, CMS) Compared against top 20 values (0.005% of Total Reads) obtained from storing in Hash Map (Naive Method).
Support (ϕ): 0.005%

Interestingly, Lossy Counting is performing better than other two methods with greater precision and recall at ε = 0.005. Whereas the other two methods have only around 60% of elements matched with the expected elements.
The following is the graph representing the number of false positives, false negatives and Actual Matched number of elements with the changing ε (error rate) using Lossy Counting Algorithm:



It is clear from the graph that as we increase **ε** (error rate), we are moving towards the worst-case performance with increasing number of false positives, false negatives and less number of expected elements. This is expected since the amount of trace (i.e. the bucket size) being kept is inversely proportional to **ε** (error rate). As number of elements in the bucket are limited, it will flush out even the MFA elements at each bucket boundary and it will cause less accurate results.

Step-2: Computation of overall execution time for each implementation to return the MFA elements.

We tested the overall execution taken by LC and SSL against the following Workload:

Scrambled Zipfian Distribution (Skew factor: 0.99)

Reads: 10,000
Key Range: 1,000
Data Size: 1,000,000
With Changing Support (ϕ)

MFA implementation Used	Time taken (in microsecs) at Support (ϕ) = 0.005	Time taken (in microsecs) at Support (ϕ) = 0.02
Lossy Counting	23132	19879
Space Saving using Lists	20432	17891

It is clear from the above table that SSL is taking less time to compute the MFA elements for the given Support (ϕ) as the only overhead is to maintain the minimum frequent element at any time and replace it with the new incoming data stream values.

Conversely, the Lossy Counting algorithm will iterate over all the elements in the bucket at the bucket boundaries. Later, it checks if the element is most frequent. If not, it will remove that element from the bucket.

Step-3 is similar and as specified earlier, the amount of space it takes to calculate the MFA elements is inversely proportional to error rate (**ε**). Size of traces the algorithms that are required to maintain is almost similar for both LC and SSL.

From the above comparison results, we can conclude that the Lossy Counting algorithm performs well with respect to precision and recall but the Space Saving algorithm has slight edge over LC with regards to time complexity. But as LC gives more predicatable results and has very less difference with respect to time taken to calculate MFA elements. Finally, LC has been integrated to the current implementation.

The integration of Lossy Counting algorithm is straightforward. A member variable named "***frequencyCounter***" has been added to the CrackingMode class in the code. Every time a new element is read, this element is added to the ***frequencyCounter***. ***frequencyCounter*** will run LC algorithm to accumulate the read counts as <ReadElement, Count> pairs in buckets. If we need read statistics, at any point of time, we simply need to call getFrequentItems(support) or getFrequentItems(<AnyNumber>) to retrive the MFA elements using support or the given number.

4.4. Tester / Utility Methods

It is commendable that a wonderful demo interface has been provided to visualize the run times using bar graphs and to visualize the running cog structure. But during the course of the project, we needed some more test/utility methods for the following reasons:

- To replicate the performance plots/graphs for cracking, adaptive merging, swap policies in current implementation.
- To identify the pain points in the current implementation for different workloads.
- To compare the run times for the implementation with and without splaying under same experimental conditions.

Therefore, for plotting graphs by parsing generated log file during execution, python scripts have been written that can provide scatter plots and line plots.

We have created a new Key-Value iterator named **ZipfianIterator** that extends **KeyValueIterator** class. It calls the Zipfian generator to get the key-value pairs on each *next*() call based on the Zipfian distribution.

```
public static class ZipfianIterator extends KeyValueIterator {
	Integer max;
	long key, value;
	ZipfianGenerator zipfRand = new ZipfianGenerator();
```

A utility method has been provided that can compare the performance of current implementation with and without splaying under same experimental conditions like same distribution values, same sequence of reads, etc. There is another utility method that takes ".sim" files as input then performs multiple runs and provides the aggregated (averaged) result of multiple runs of each ".sim" file.

5. Experimental Analysis

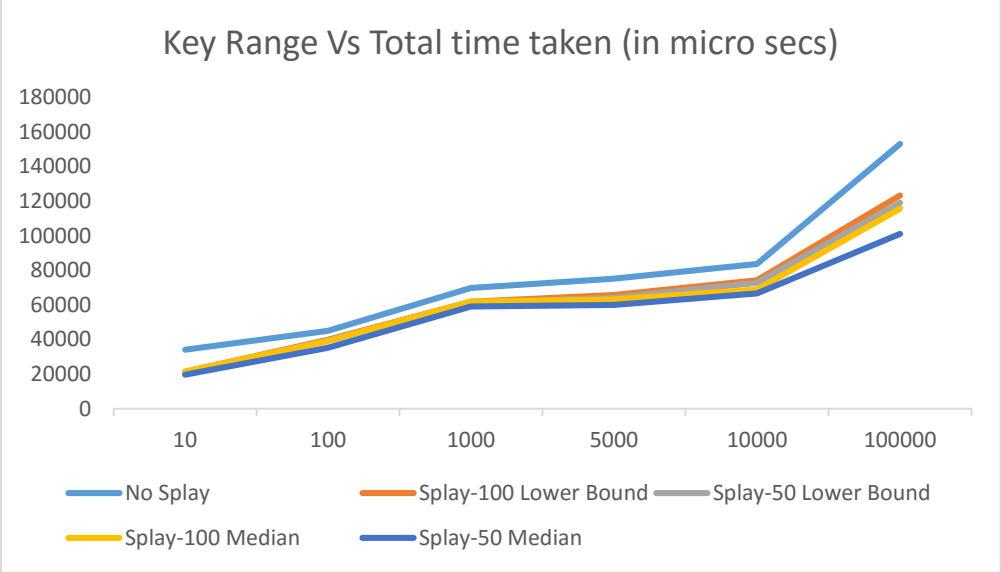
Experimental Setup:

All the experiments performed for the testing of the policies from the beginning till date have been implemented over the below given setup:

Processor	:	Intel Core i5 2.40 GHz
Memory	:	8 GB DDR3 RAM
Operating System	:	Ubuntu 14.10 and JDK 1.7
JVM parameters	:	Heap Space = 5 GB, Stack Space = 200 MB

5.1. Performance of splaying on Median vs. Lower Bound

The following is the graph plotted between Key Range and the total time taken for 1000 reads on implementation without splaying, with splaying on each 100 and 50 reads using lower bound of range query as element to splay and using median as splay element. We used data of size 1000000 following Zipfian distribution in cracker mode with read width 1000. Each observation is the aggregated value of 5 runs. Please note that in the below graph we haven't considered the time taken to calculate the median values.



It is clear from the graph, for the varying Key ranges, splaying actually improves the performance, splaying around median outperforms splaying around lower bound and implementation without splaying. Since splaying around median balances the tree better with even distribution of underlying sub-array and leaf cogs. Even though results are satisfying there is variance between the runs which is expected due to caching effect between the runs.

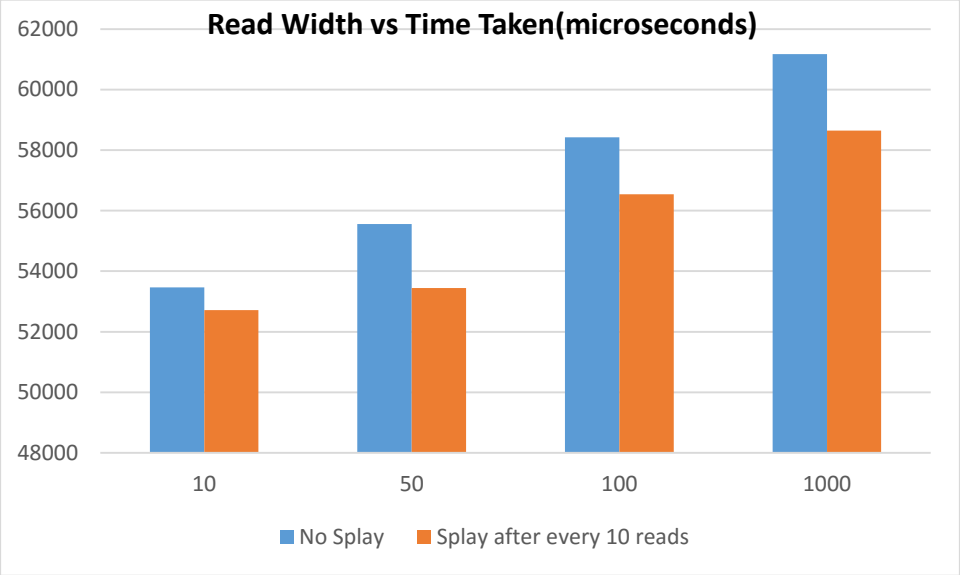
The following sections provides the plots for splaying around median for varying Key Range, Read Width and Splay Intervals.

5.2. Performance of splaying on Median on other parameters

We have performed the splay operations taking into consideration of various parameters to check how each of them causes a difference in the implementation, and then come up with the best approach (or parameter values) so that we get an optimal performance over the series of runs. We have tested the workload by changing the read width, key range as well as the splaying interval. Detailed analysis is given below.

5.2.1. Read Width

Experiments on the read width have been done for several values of *10, 50, 100, 500, 1000* and we have checked the implementation behavior over the Zipf workload with and without splay operation.

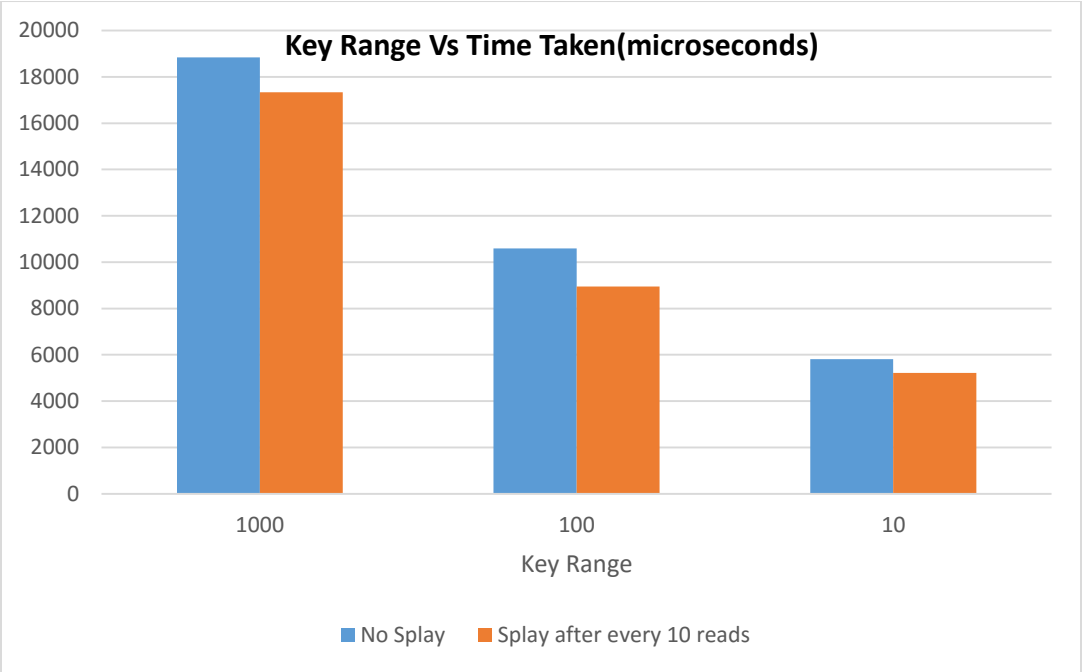


All the displayed results are based on the time taken (in microseconds) for *1000* reads and is aggregated over 5 runs on each type. We used the data of size *1000000* following the Zipfian distribution in cracker mode with a constant key range *1000*.

The plot shows even with varying Read Width splaying after every 10 reads shows performance improvement.

5.2.2. Key Range

Similarly, experiments on the key range have been done for several values of *10, 100, 1000* and we have checked the implementation behavior over the Zipf workload with and without splay operation.



All the displayed results are based on the time taken (in microseconds) for *1000* reads and is aggregated over 5 runs on each type. We used the data of size *100000* following the Zipfian distribution in cracker mode.

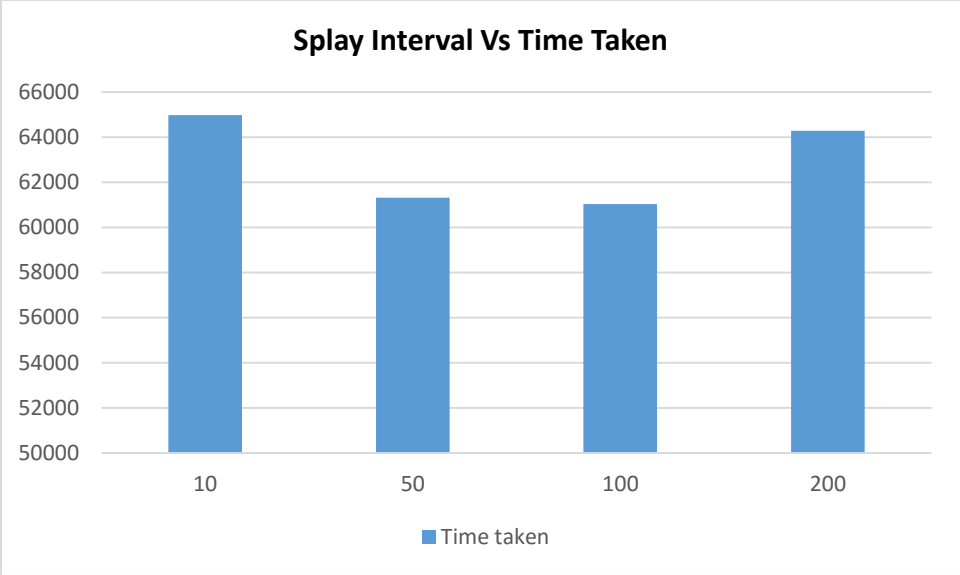
Less Number of Keys implies less number of keys to be cracked and it takes less amount of time for all reads. Even with splaying it holds true with extra performance benefit due to property of tree balancing.

5.2.3. Splaying Interval

Again, similar experiments have been done for several values of the splaying intervals - 10, 50, 100, 200 and 500, and we have checked the implementation behavior over the Zipfian workload.

All the displayed results are based on the time taken (in microseconds) for *1000* reads and is aggregated over 5 runs on each type. We used the data of size *100000* following the Zipfian distribution in cracker mode with a constant read width *1000* and a constant key range *1000.*

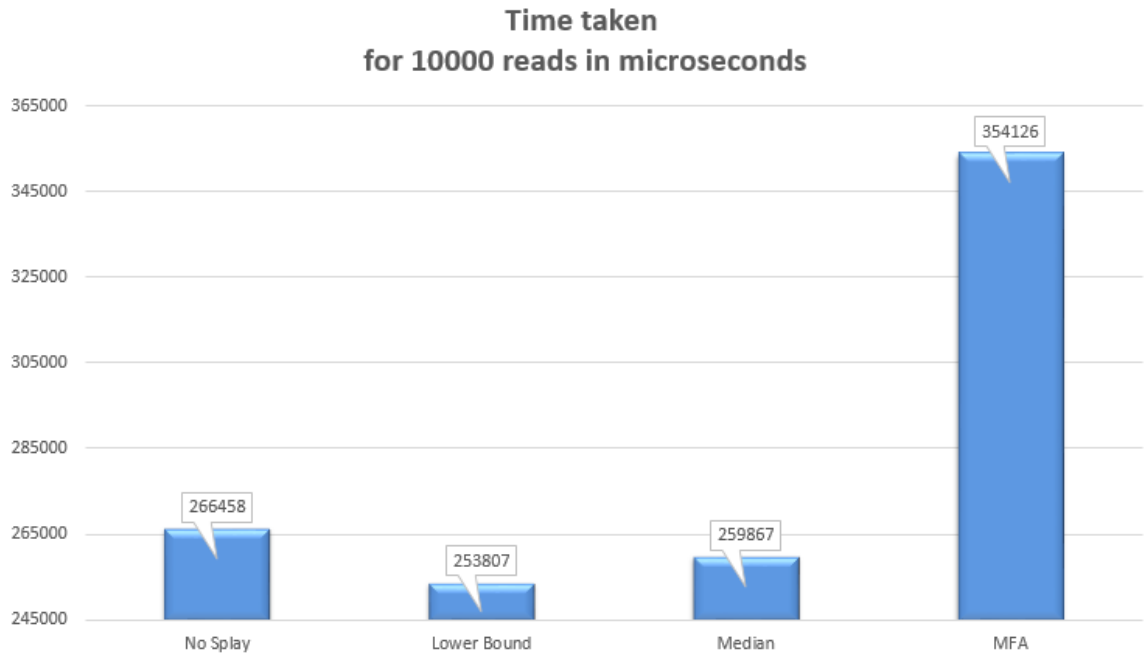
It is evident from the graph that more splaying is an overkill and can actually degrade the performance. As of now, from our experimentation splaying after every 50 or 100 reads is reasonable.



5.3. Overall Performance Comparison

We have performed experiments to calculate the overall cost for all the 4 policies that have been discussed till here. We have used the data size of 1,000,000 following Zipfian distribution in Cracker mode with a constant key range of 100,000.

In the below graph, the results have been plotted as the type of policy against time taken (in microseconds) for each of them that includes the time required for auxiliary operations like calculating the median element or generating the most frequently accessed elements. Splay interval used for each policy is 100.



It is clear from the graph that splaying over the lower bound of the range query performs optimally while compared to splaying on median or most frequently accessed elements. Splaying over the median adds the cost of calculating the median value in the JITD, whereas splaying over the MFA adds the cost of collecting user statistics from the queries and aggregating the results to generate a list of MFA elements. Thus, the policy of splaying on lower bound occupies the first place in obtaining the best results based on our experiments.

For the same setup, one interesting observation here is regarding the splay operation on median elements. The below scatter plots show the comparison between the experiments done to compare the performance of triggering 10000 reads with and without splaying on median elements.

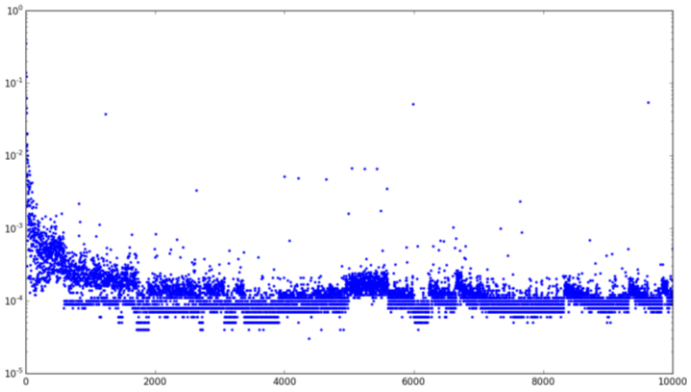


Figure 5.3.1: No splay (10000 reads)

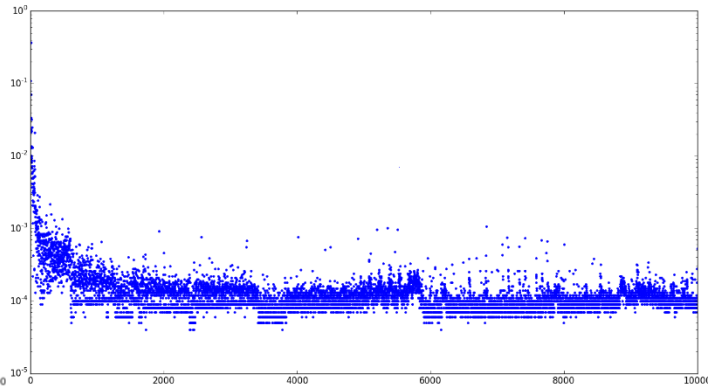


Figure 5.3.2: Splay at Median (10000 reads)

We can see that the splay operation on median element actually smoothens the curve as the balancing on the median element divides the entire data structure into two equal halves thereby reducing the access time of elements which have low selection probability.

6. Future Work

6.1. Exploring Policy for Splaying at variable intervals of Time

Some of the ideas under consideration for splaying at variable intervals of time are:

- Using Sedgewick's gap sequence formula to generate gaps using geometric progression. Gaps can be taken as splay intervals for our purpose.

  $$4^k + 3 \cdot 2^{k-1} + 1, prefixed\ with\ 1 \quad generates\ \ 1,8,23,77,281 \ldots\ldots\ldots$$

- Another idea is to slow down the splay process if all the most-frequent elements are closer to the root cog with a maximum of 'k' levels.
- Idea of variable intervals can even exploit the rate at which new cogs (separators) are being created. For example, for initial reads, rate of creation of new separators (cogs) is more, so there is more chance for splaying which indicates that the splay operation can be more often around median whereas it can be less often while reaching convergence which can be estimated based on (*Number of new Separators created/Number of Reads*) ratio.

6.2. Hybrid policy based on varying splay elements:

As observed from the experiments that splaying on lower bound and median outperformed other policies. We can design a hybrid policy that initially splays on lower bound when the number of nodes are less. After around 1000 reads, splaying on median will nearly balance the JITD structure. After retrieving sufficient amount of metadata required for MFA, we can perform splay on Most Frequently Accessed element without the need for recalculating the user stats, so that most frequently accessed element will be on top of tree and the other elements will be accessed in a better response time.

7. Conclusion

We have observed that splaying reduces the height of JITDs tree structure through balancing which in-turn reduces the time to lookup elements for the range query. But at the same time the re-organization of the structure doesn't come without cost.

We have experimented varying read intervals for performing splay operation over the existing JITD structure and came to a conclusion that splaying after every 100 reads performs in optimal time.

All the policies have been tested under identical experimental setup and it was established that the policy of JITDs with splaying on lower-bound of the range query outperformed the others. This can be attributed to the fact that splaying on lower bound requires no auxiliary operations as compared to the cases of splaying on median element (or) most frequently accessed elements. Having this inferred we could say that this work could be extended further in order to improve the policies as per some ideas mentioned in future work.

8. References

1. An improved data stream summary: the count-min sketch and its applications by Graham Cormode,S. Muthukrishnan. Journal of Algorithms Volume 55, Issue 1, April 2005, Pages 58–75.
2. Approximate frequency counts over data streams by Gurmeet Singh Manku, Rajeev Motwani. VLDB '02 Proceedings of the 28th international conference on Very Large Data Bases, Pages 346-357.
3. Just-In-Time Data Structures by O. Kennedy, L.Ziarek. CIDR 2015, Paper 9.
4. Metwally, Ahmed, Divyakant Agrawal, and Amr El Abbadi. "Efficient computation of frequent and top-k elements in data streams." Database Theory-ICDT 2005. Springer Berlin Heidelberg, 2005. 398-412.
5. Methods for finding frequent items in data streams by Graham Cormode, Marios Hadjieleftheriou. The VLDB Journal Volume 19 Issue 1, February 2010 , Pages 3-20.
6. A collection of algorithms for mining data streams https://github.com/mayconbordin/streaminer