Poster Paper

SPST-Index: A Self-Pruning Splay Tree Index for Caching Database Cracking

Pedro Holanda
UFPR, Brazil
pttholanda@inf.ufpr.br

Eduardo Cunha de Almeida
UFPR, Brazil
eduardo@inf.ufpr.br

ABSTRACT

In database cracking, a database is physically self-organized into cracked partitions with cracker indices boosting the access to these partitions. The AVL Tree is the data structure of choice to implement cracker indices. However, it is particularly cache-inefficient for range queries, because the nodes accessed only for a few times (i.e, "Cold Data") and the most accessed ones (i.e, "Hot Data") are spread all over the index. In this paper, we present the Self-Pruning Splay Tree (SPST) data structure to index database cracking and reorganize "Hot Data" and "Cold Data" to boost the access to the cracked partitions. To every range query, the SPST rotates to the root the nodes pointing to the edges and to the middle value of the predicate interval. Eventually, the most accessed tree nodes remain close to the root improving CPU and cache activity. On the other hand, the least accessed tree nodes remain close to the leaves and are pruned to improve updates. Our experimental evaluation shows 37% more Instructions per Cycle and 75.9% less cache misses in L1 for lookup operations in the SPST compared to the AVL tree. Our data structure outperforms the AVL tree for lookups and maintenance costs in three major data access patterns: random, sequential and skewed. The SPST outperforms the AVL in 4% even in the worst case scenario with mixed workloads with lookups and batch updates.

Keywords

Database Cracking, Cracker Index, Splay Tree

1. INTRODUCTION

Database Cracking [6] presents a self-organizing database partitioning for column-oriented relational databases. It works by physically self-organizing database columns into partitions, called cracked pieces. The goal is to create cracked pieces for all accessed intervals of range queries. Cracker indices are created to keep track of these partitions.

An index is a data access method that typically stores a list of pointers to all disk blocks that contain records to the indexed column. The values in the index are ordered to make binary search possible. It is smaller than the data file itself, so searching the index using binary search is reasonably efficient [2]. In contrast to usual indices in the literature, the nodes of a cracker index do not point to all the disk blocks of a column. Instead, they point to the beginning of each cracked piece to boost access to an interval of values.

The current data structure implemented as cracker index is the self-balancing AVL Tree [1], where the height of the adjacent children subtrees of any node differ by at most one. If in a given moment their height differs by more than one, the tree is rebalanced by tree rotations. As the index is created by incoming queries, the index starts to be filled with pointers to data keeping the self-balancing property of the tree height. However, this property makes the AVL tree particularly cache-inefficient. The tree nodes accessed only for a few times (i.e, "Cold Data") and the most accessed ones (i.e, "Hot Data") are spread all over the index. Another concern lies in the index size as "Cold Data" are kept in the index. Eventually, the cracker index converges to a full index (i.e, all values indexed) with high administration costs for high-throughput updates.

In this paper, we present a data structure called Self-Pruning Splay Tree (SPST) to index database cracking and keep "Hot Data" close to the root of the tree. The SPST is based on binary search Splay trees with a self-adjusting property carried out by the *splaying* operation. *Splaying* consists of a sequence of rotations to move a node way up to the root of the tree. To every range query, our algorithm rotates the nodes pointing to the edges and to the middle value of the predicate interval. With "Hot Data" constantly rotated, they eventually remain close to the root. On the other hand, "Cold Data" are stored close to the leaves presenting the opportunity to prune them out of the index and improve maintenance and update costs.

This paper is organized, as follows: Section 2 discusses related work. Section 3 presents our Cracker Index followed by the experiments in Section 4 and we conclude in Section 5.

2. STANDARD DATABASE CRACKING AND RELATED WORK

There are two Database Cracking algorithms: *crack-in-two* and *crack-in-three* to split the columns into two and three partitions respectively. The first one is suited for one-sided range queries (e.g, $V_1 < A$) or two-sided range queries (e.g, $V_1 < A < V_2$) where each side accesses different cracked pieces. The second one is only for two-sided queries that access the same cracked piece. It starts with similar per-

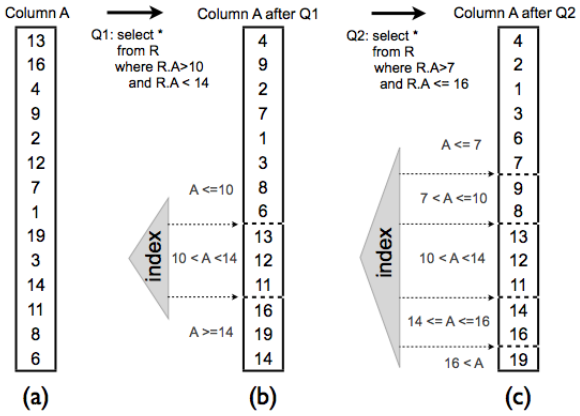


Figure 1: Database Cracking when executing two queries with different ranges [8]

formance of full column scan and overtime gets close to the performance of a full index.

Figure 1 depicts query Q_1 triggering the creation of the cracker column A_{ckr}, (i.e., initially a copy of column A) where the tuples are clustered in three pieces reflecting a *crack-in-three* iteration from the range predicate of Q_1. The result of Q_1 is then retrieved as a view on Piece 2 (i.e., indexing $10 < A < 14$). Later, query Q_2 requires a refinement of Pieces 1 and 3 (i.e., respectively indexing $A > 7$ and $A \leq 16$), splitting each in two new pieces resulted by a *crack-in-two* iteration.

There are many data structures in the literature to keep track of data partitions. In database cracking the AVL is the data structure of choice, but other self-balancing trees, like RedBlack or 2-3 trees, draw the same result. These trees have the property of keeping the height of the tree for self-balancing purposes. However, this property makes them cache-inefficient for range queries. The tree nodes accessed only for a few times and the most accessed ones are spread all over the tree.

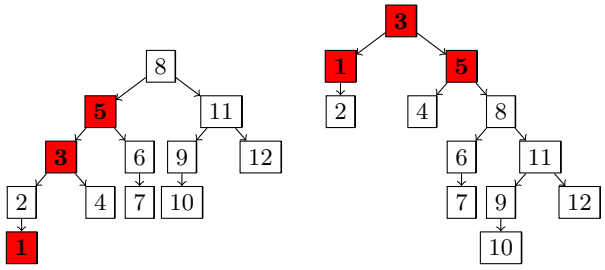
3. THE SPST-INDEX

Our contribution regards recognizing "hot data" to improve data access and recognizing "cold data" to prune unused data and boost updates.

3.1 Splaying

A Splay Tree [9] is a self-adjusting binary search tree that uses a splaying technique every time a node is Searched, Updated, Inserted or Deleted. *Splaying* consists of a sequence of rotations that moves a node to the root of the tree. Lookup, Insertion and Deletion take $O(\log n)$ time in the average and worst case scenarios, where n is the number of nodes in the Splay Tree. It clusters the most accessed nodes near the root of the tree. Therefore, the most frequent accessed nodes will be accessed faster. Since we are dealing with range queries, our goal is to splay the query range, instead of splaying only one node like the original splay tree. The self-adjustment algorithm in our data structure is straightforward: we first splay the leftmost node of the range, then the rightmost node and later the closest node to the middle.

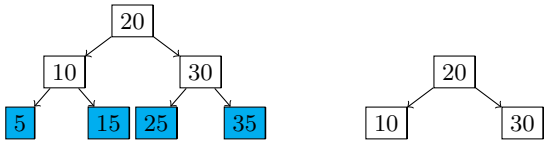
Let us consider for cracker index the SPST depicted by Figure 2. If a range query of $1 < A < 5$ is executed, the



(a) The SPST before the incoming query $1 < A < 5$

(b) The SPST after Splaying nodes 1, 5, and 3

Figure 2: The SPST splaying the range $1 < A < 5$



(a) The SPST before pruning the leaves

(b) The SPST after pruning the leaves

Figure 3: The SPST with size $n = 7$ being pruned.

algorithm performs three operations: Splay (1), Splay (5) and Splay ($\lceil \frac{(1+5)}{2} \rceil$). Figure 2(b) depicts the resulting tree with nodes 1, 3 and 5 close to the root. In the SPST, the nodes remain close to the root as long as they are frequently accessed. In our index, the nodes pointing to the most accessed cracked pieces remain close to the root.

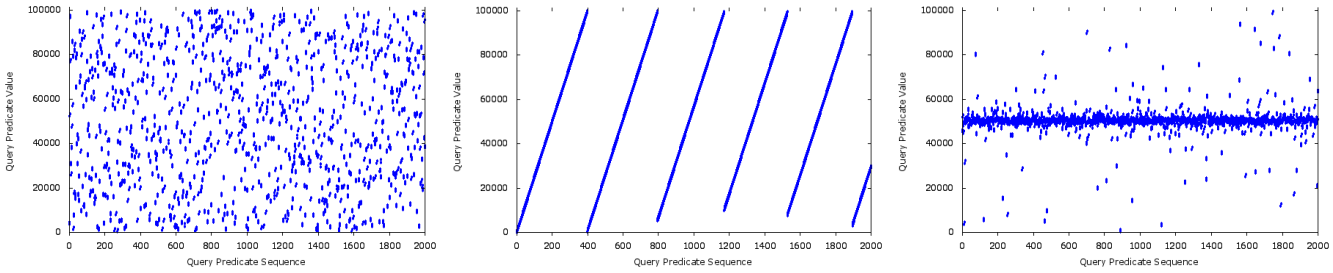
3.2 Pruning

Besides speeding up the access to hot data, another goal is to speed up updates and maintenance costs when rotating hot data. We assume that eventually the nodes stored at the leaves point to cold data. The maintenance strategy of our data structure is to prune the leaves. As we prune them, the update time is expected to shrink. The downside of pruning the tree is that the following queries can become slightly more expensive compared to the situation where we do not have any pruning at all. Our hypothesis is that we mitigate this cost with the gains in the update time. When we prune the leaves, the size of the index shrinks, in the best case, to $\lfloor \frac{n}{2} \rfloor$, where n is the number of nodes in the SPST.

Let us suppose the SPST index depicted by Figure 3(a). In this scenario the most frequent range is between 10 and 30. Let us suppose inserting the value 21 in the Cracker Column. To do this, we need to update the nodes 35, 30 and 25 respective pointers to the cracker column and merge at their respective cracker column pieces. Instead, we start pruning the leaves having as result the tree depicted by Figure 3(b). Then we only need to update the pointer to the cracked piece of node 30.

4. EXPERIMENTAL ANALYSIS

In this section, we discuss the results of our experimental evaluation of the SPST implemented as a cracker index. We divide this section in two subsections, the first one is related to the select operator where we performed the same



(a) Random (b) Sequential (c) Skewed

Figure 4: Workload Patterns

experimental protocol and ran the same lookup scenarios described in [8]. The second one is related to the update scenario where we performed the same experimental protocol and ran the same scenarios described in [4]. We implemented our data structure and performed all the experiments using the database cracking simulator[1] presented by [8]. We ran the experiments on a MacOS Sierra (10.12) machine with 2.2GHz quad-core Intel Core i7 processor (Turbo Boost up to 3.4GHz), 6MB shared L3 cache and 8 GB of RAM.

For the select operator, we focused our analysis on the accumulated index lookup time for querying and indexing, and the accumulated index update time. In particular, we analyzed the Instructions per Cycle (IPC) and the cache misses (L1/2/3). We consider as the best cache-efficient data structure the one with the highest IPC and lowest number of cache misses.

For the update operator, we considered two update scenarios: low frequency high volume updates (i.e, LFHV), and high frequency low volume updates (i.e, HFLV). In the first scenario after 1,000 queries a batch of 1,000 updates are executed. In the second scenario after 10 queries a batch of 10 updates are executed. The query pattern and the updates are both random. The SPST prunes itself always before a batch update if the previous queries present a standard deviation, for cracking time, lower than a defined threshold. We focused our analysis only on measurements that are affected by update and pruning (i.e, cracking time, index update time, cracker column shuffle time and pruning time).

We use an integer array with 10^8 uniformly distributed values. The workload size and the query selectivity is 1,000 and 1 for all experiments. All query predicates are of the form: $R.A \geq V_1$ AND $R.A < V_2$. We repeat the entire workload 5 times and take the average runtime of each query. We consider three different workloads depicted by Figure 4. For each workload, we graphically illustrate how a sequence of $1,000$ queries accesses the domain value of a single attribute. For each query, we plot the two edges of the interval (i.e., called "Query Predicate Sequence"). The random, sequential and skewed workloads are respectively depicted by Figures 4(a), 4(b), and 4(c). The skewed workload is generated by the zipf's law with α equals to 2.0.

4.1 Select Operator

Figure 5 depicts the accumulated index lookup and maintenance time for the query stream in the random, sequential and skewed workload. For random, the AVL Tree was faster than the SPST for the first 180 queries, because the random workload demanded a higher number of rotations in the SPST to settle down the range pattern close to the root. With more incoming queries the SPST started to leverage the cached nodes from the root running the $1,000^{th}$ query 21.5% faster than the AVL Tree (see Figure 5(a)).

The sequential pattern was the worst case scenario for the SPST, but still the SPST was 7% faster than the AVL Tree at the $1,000^{th}$ query (see Figure 5(b)). The worst case scenario was the result of many changes in the range predicate of the sequential pattern that required splaying many nodes from the leaves. Over time the SPST mitigated these rotations with 16.9% less cache misses compared to the AVL (see Table 1). The skewed pattern was the best case scenario for the SPST, being 37% faster than the AVL Tree at the $1,000^{th}$ query (see Figure 5(c)). The best case scenario was the result of a skewed workload, achieving an IPC 37% higher. (see Table 1).

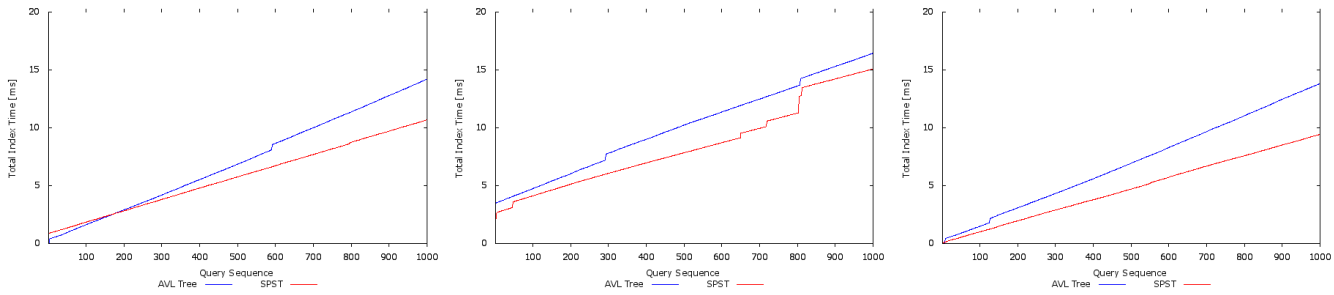
Tree	L1	L2	L3	IPC
Random				
AVL	1108508606	4972838130	252404784	1.094
SPST	267097844	3957313535	135615510	1.385
Sequential				
AVL	855925856	10890330930	412469096	1.234
SPST	711228747	10479242239	399344564	1.263
Skewed				
AVL	573854301	3800678199	176536452	1.160
SPST	256760334	3780063118	128213328	1.600

Table 1: Cache Misses and IPC by workload

4.2 Update Operator

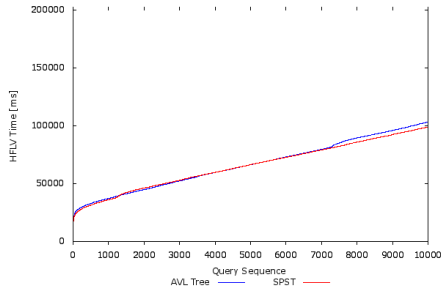
Figures 6(a) and 6(b) depicts the accumulated cracking and update time for the query stream of $10,000$ queries in the HFLV and LFHV scenarios respectively. In both, the SPST achieves the lowest run time. Every time the tree is pruned, updates are boosted but cracking becomes more expensive since we have less nodes to update, but bigger pieces of the cracker column to scan. The SPST was able to prune at convenient moments minimizing the extra cracking cost and greatly boosting update time. For HFLV, we defined empirically a standard deviation of 0.2 milliseconds and for LFHV 200 milliseconds. These values differ because for HFLV it is only analyzed the standard deviation for 10 queries previous to a batch update, while for LFHV 1,000 queries are analyzed. For HFLV, the SPST was pruned only

[1]The cracker index simulator, written in C/C++ and compiled with G++ v.4.7, is available at: www.infosys.uni-saarland.de/research/publications.php

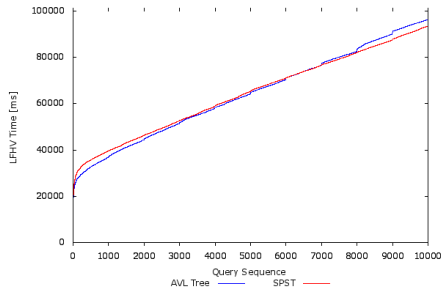


(a) Random Workload (b) Sequential Workload (c) Skewed Workload

Figure 5: Sum of Lookup for Querying and Indexing, and Insertions Time in Various Workloads



(a) HFLV



(b) LFHV

Figure 6: Total time for cracking and updates

once, and was 4% faster than the AVL Tree. For LFHV, the SPST was 5% faster, pruning the tree 8 times and having around 25% of the total size of a full AVL Tree. We observed more rotations in the SPST than in the AVL tree. However, the rotations in the SPST presented less impact in response time compared to the ones in the AVL Tree. While in the SPST the rotations happened most frequently near the root with less cache misses in L1/2/3 and higher IPC, the AVL Tree spanned many rotations usually close to the leaves of the index to rebalance the tree with many unnecessary tree nodes polluting the cache (see Table 1 cache misses).

5. CONCLUSION

This work presented the SPST as a cracker index for database cracking. We explored the Standard Cracking algorithm for select and mixed workloads with three different synthetic patterns where the SPST outperforms the AVL Tree in all scenarios. The SPST was able to cache the most frequently accessed data near to the root reducing cache misses and achieving a higher IPC than the AVL.

In future work, we will compare the SPST with other main-memory index structure for efficiently executing queries on modern processors, like, the recent proposed ART-Tree and Cache-Sensitive Skip List. Our SPST implementation follows the classic splay tree structure, but there are many rotation and pruning strategies that can be explored to improve response time, like, freezing the top of the SPST to diminish the rotations. Furthermore, we focus the SPST on standard cracking. However, there are other cracking approaches in the literature to be explored in future work, like: Hybrid Cracking[7], Sideways Cracking[5] and Stochastic Cracking[3].

Acknowledgments

This work was partly funded by the CNPq Universal, grant 441944/2014-0.

6. REFERENCES

[1] J. Bell and G. Gupta. An evaluation of self-adjusting binary search tree techniques. *Software: Practice and Experience*, 23(4):369–382, 1993.

[2] Elmasri and Navathe. *Fundamentals of Database Systems*. Pearson, 2007.

[3] F. Halim, S. Idreos, P. Karras, and R. H. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *VLDB*, 5(6):502–513, 2012.

[4] S. Idreos, M. L. Kersten, and S. Manegold. Updating a cracked database. In *SIGMOD*, pages 413–424, 2007.

[5] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. *SIGMOD*, pages 297–308, 2009.

[6] S. Idreos, M. L. Kersten, S. Manegold, et al. Database cracking. In *CIDR*, volume 3, pages 1–8, 2007.

[7] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores. *VLDB*, 4(9):586–597, 2011.

[8] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The uncracked pieces in database cracking. *VLDB*, 7(2):97–108, 2013.

[9] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.