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

Pedro Holanda[1], **Eduardo Cunha de Almeida**[1]

[1]Programa de Pós-Graduação em Informática (PPGINF)
Universidade Federal do Paraná (UFPR)
Curitiba – PR – Brazil

`{pttholanda,eduardo}@inf.ufpr.br`

- Masters admission date: 02/2015
- Proposal defense date: 08/2016
- Expected finish date: 02/2017
- Concluded steps:
  - Implementation of the Select Scenario;
  - Implementation of the Self Pruning Splay Tree;
  - Results of the Select Scenario.
- Future steps:
  - Implementation of the Update Scenario;
  - Results of the Update Scenario;
  - Writing the masters dissertation.

Abstract. *A cracked database is a physically self-organized database based on the predicates of queries being executed. The goal is to create self-adaptive indices as a side-product of query processing. Our goal is to critically review the current cracking algorithms and index structures with respect to OLAP and mixed workloads. In this dissertation, we present a new data structure to keep small parts of the cracker attribute at the top of the tree to act as cache. Instead of using the current AVL Tree as a cracker index, we implement a self-pruning Splay Tree (SPST), with the objective to cluster the main accessed nodes at the root and to prune the external nodes to improve updates. The main challenge is related to caching data and reduce cost to build and maintain the indices as the current data structure is not convenient to prune unused data. Finally, we present preliminary results of our SPST implementation against the current database cracking index AVL Tree for read operations. In our preliminary results, the SPST shows better response times in* 12% *compared with the AVL Tree for Random Workloads and for Skewed Workloads SPST shows* 20% *better response time.*
Keywords: Database Cracking, Cracker Index, Splay Tree

Resumo. *Database Cracking é uma técnica que organiza fisicamente um banco de dados baseado nos predicados das consultas que são executadas, criando índices dinâmicos como resultado do processamento de consultas. Nosso objetivo é revisitar os algoritmos e estrutura de dados utilizadas para índices de Database Cracking considerando cargas de trabalho OLAP e mistas. Nossa proposta apresenta a substituição da Árvore AVL pela SPST (Estrutura baseada* *em Árvores Splay). O principal objetivo da SPST é de agrupar os nós mais acessados próximos a raiz da árvore para otimizar as operações de busca e podar os nós externos para diminuir o tempo das atualizações. O desafio é relacionado ao custo de construir e manter índices onde não se é possível excluir os dados pouco acessados. Por fim, apresentamos os resultados da nossa implementação da SPST comparando-a com a Árvore AVL. Como resultados preliminares, temos que a SPST demonstra ganhos em tempo de resposta para OLAP de 12% em padrões de acesso aleatórios e 20% em padrões de acesso enviesados.*

1. Introduction

Indices are data access methods that typically store each value of the indexed field along with a list of pointers to all disk blocks that contain records to that field value. The values in the index are ordered to make binary search possible. It is smaller than the data file, so searching the index using binary search is reasonably efficient [Elmasri and Navathe 2007].

However, creating indices is not a simple task. Knowing which index to create, how to create them, and which queries will use them is a task that requires knowledge of many parameters that change according to the workload (i.e, read and update operations). In the past few decades, some tools have been developed to make this task easier. The Self-Tuning tools are able to capture relevant workload patterns and then suggest physical design improvements to the database administrator (DBA).

Database Cracking [Idreos et al. 2007b] has been proposed for column-oriented relational databases to create self-organizing databases. In this way the workload does not have to be known a priori. Database Cracking works by physically self-organizing database columns into pieces, called cracked pieces, and generating an index to keep track of those pieces.

For example, assuming the following predicate $A < 10$. The idea of database cracking is clustering all tuples within $A < 10$ in the beginning of the column and pushing the remaining tuples to the end. As so, incoming queries with predicates $A > V1$, where $V1 \geq 10$, will only look up the second part of the column. A cracker index maps the column positions cracked so far, allowing the query processor to take advantage of it. The more queries are processed, the more the database is partitioned into smaller and manageable pieces making data access significantly faster [Idreos et al. 2007b].

The current data structure used as a Cracker Index is an AVL Tree, which is a self-balancing binary search tree where the height of the adjacent children subtrees of any node differ by at most one. As the index is created by incoming queries, it will converge to a full index, bringing new issues, such as higher administration costs for high-throughput updates (i.e, going through the height of a tree). They happen because the index starts to be filled with unused data that is data only retrieved a few times (i.e, "Cold Data"). Our goal with the cracker index is to store the most accessed nodes (i.e, "Hot Data"), near the root of the tree. For this objective Splay Trees are a better fit than AVL Trees.

A Splay Tree is a self-adjusting binary search tree which 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. Our goal with splay tree in database cracking is to make the index comply with the cracking philosophy, by always adapting itself to the incoming workload. This is achieved by clustering hot data at the root of the tree to boost data access. On the other hand, the least accessed nodes are stored at the leaves, giving the opportunity to prune cold data out of the index and boost further update operations.

This dissertation proposal aims to contribute with the following:

- A data structure inspired by Splay Trees to keep hot data at the top of the tree to improve read operations.
- A strategy to prune cold data kept in the external nodes of the tree to improve write operations.

This paper is organized, as follows: Section 2 discusses related work. Section 3 presents our proposal. Section 4 depicts our preliminary results. Finally, section 5 discusses future work.

2. Related Work

In the following section we present a general view of Cracker Index Data Structures and Cracking Algorithms.

2.1. Cracker Indices

The current data structure used as a Cracker Index is an AVL Tree [Idreos et al. 2007b], which is a self-balancing binary search tree where the height of the adjacent children subtrees of any node differ by at most one [Bell and Gupta 1993]. If in a given moment they differ by more than one, the tree is rebalanced by tree rotations. Lookup, Insertion and Deletion take $O(log\ n)$ time in average and worst cases, where n is the number of nodes in the AVL tree.

2.2. Database Cracking Select

Cracking is done by two algorithms: *crack-in-two* and *crack-in-three*, which split the column 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 touches different cracked pieces. The second one is only for two-sided queries that touch the same cracked piece. It starts with similar performance of full column scan and overtime gets close to the performance of a full index. Our Database Cracking implementation sticks to these algorithms [Idreos et al. 2007b]. Although, different cracking approaches exist, as follows:

- Hybrid Cracking: Create unsorted initial pieces which are physically reorganized and then adaptively merged for faster convergence to a full index. Created to address the issue of poor convergence of standard cracking into full index [Idreos et al. 2011].
- Sideways Cracking: Adaptively creates, aligns, and cracks every accessed selection-projection attribute pair for efficient tuple reconstruction. Created to address the issue of inefficient tuple reconstruction in standard cracking [Idreos et al. 2009].

- Stochastic Cracking: Creates more balanced partitions using auxiliary random pivot elements for more robust query performance. Created to address the issue of performance unpredictability in database cracking [Halim et al. 2012].

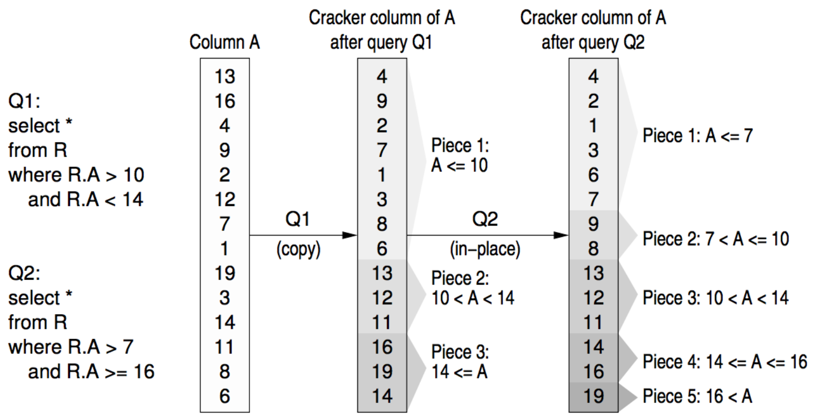


Figure 1. Database Cracking when executing two queries with different ranges [Idreos et al. 2007b]

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 resulted by a *crack-in-three* iteration, reflecting the ranges defined by the predicates. The result of Q_1 is then retrieved as a view on Piece 2. Later, query Q_2 requires a refinement of Pieces 1 and 3, splitting each in two new pieces resulted by a *crack-in-two* iteration.

2.3. Database Cracking Update

There are two basic structures to consider for updates in Database Cracking, the cracker column and the cracker index [Idreos et al. 2007a]. A Cracker Index I maintains information about the cracked pieces of a Cracker Column C. So, if a new tuple is inserted, deleted or updated in any position of C, we must update the same information into I. In Database Cracking, updates require an additional data structure, called pending insertion column, to avoid contention problems.

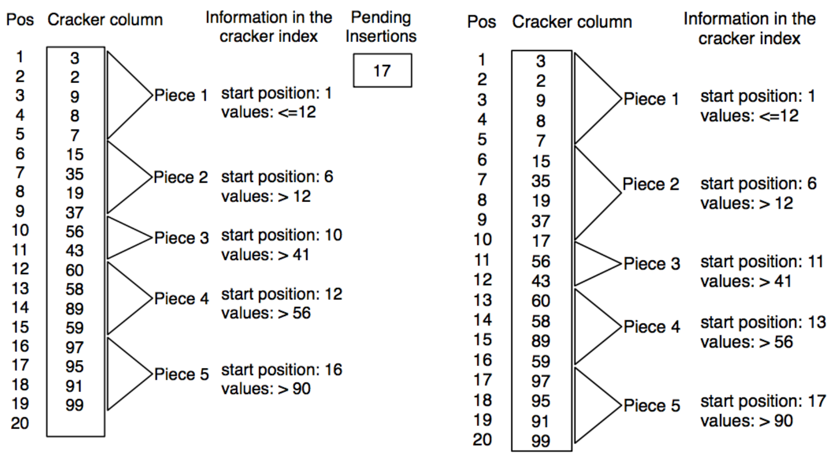


Figure 2. Updating a Cracked Column [Idreos et al. 2007a]

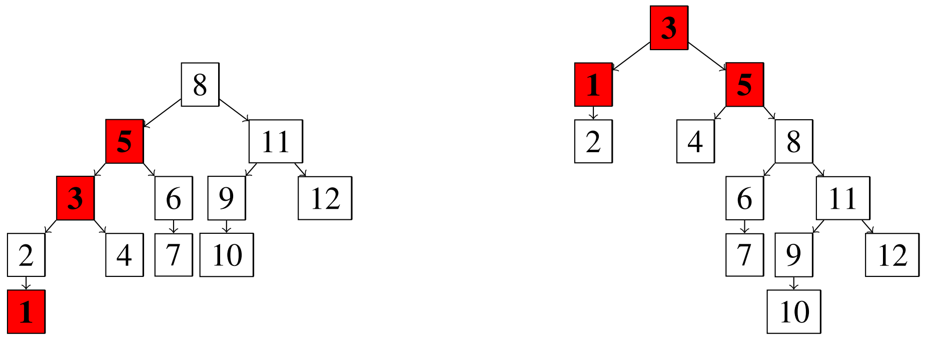
"The left-hand part of the figure depicts a cracker column, the relevant information kept in its cracker index, and the pending insertions column. For simplicity, a single pending insert with value 17 is considered. Assume now a query that requests $5 < A < 50$, thus the pending insert qualifies and should be part of the result. In the right-hand part of the figure, we see the effect of merging value 17 into the cracker column. The tuple has been placed in the second cracked piece, since, according to the cracker index, this piece holds all tuples with value v, where $12 < v \geq 41$. Notice, that the cracker index has changed, too. Information about Pieces 3, 4 and 5 has been updated, increasing the respective starting positions by 1." as described by [Idreos et al. 2007a].

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

Our first contribution regards recognizing hot data to improve data access. Our goal is to drop the index advice for cold data in the database speeding up updates. This is achieved by replacing the AVL Tree for a Data Structure inspired by Splay Tree called SPST-Index: A Self Pruning Splay Tree Index for Database Cracking as a Cracker Index. Our hypothesis is that Splay Trees are a better fit Data Structure for Database Cracking than AVL. They differ in several aspects, the main one is that the SPST clusters the most accessed nodes near to the root, allowing faster access and straightforward pruning, since we can preserve the interval nodes that are more relevant to our workload.

3.1. Splaying Strategy

A Splay Tree [Sleator and Tarjan 1985] 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. The SPST 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, it becomes necessary to find a way to splay the range, instead of splaying only one node like the original splay tree. Our algorithm is straightforward, we first splay the leftmost node of the range, then the rightmost node and the closest node to the middle. As an example, let us consider the following SPST as a cracker index:



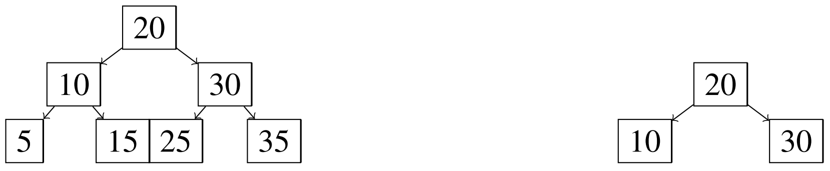
(a) Original Tree (b) Tree After Splaying 1,5,3

Figure 3. SPST with query 1 < A < 5

If a range query of $1 < A < 5$ is executed, we do three splay operations, Splay (1), Splay (5), Splay ($\lceil \frac{(1+5)}{2} \rceil$). Figure 3(b) depicts the resulting tree. We can see the nodes 1, 3 and 5, in red, are clustered near the root and remain there as long as they are frequently accessed.

3.2. Pruning Strategy

One of our main goals is to speed up updates by dropping the index reference to cold data. We assume that the nodes stored at the leaves, are the ones that are pointing to cold data. As we prune the leaves, the update time is expected to shrink with minimal lose at the select time, also when we prune the leaves the SPST shrinks to $\lfloor \frac{n}{2} \rfloor$, where n is the number of nodes in the SPST. The SPST follows a naive approach, always pruning before further updates. Let us suppose we have the SPST in Figure 4(a) as the cracker index:



(a) SPST before Pruning (b) SPST after Pruning

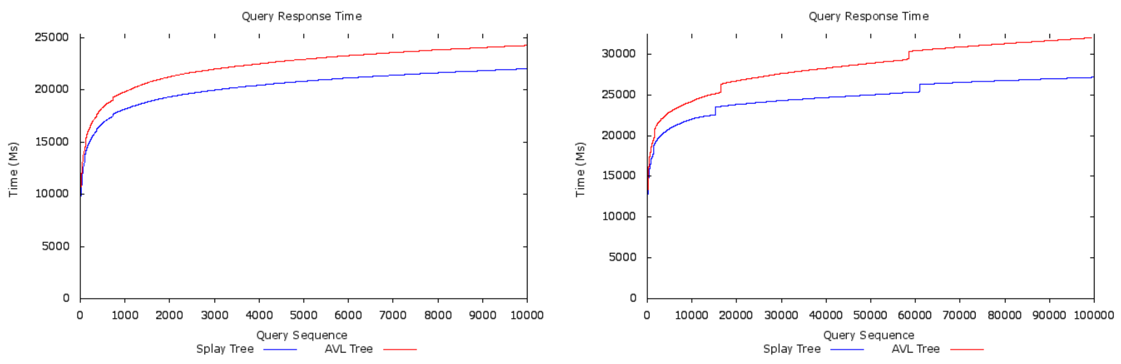
Figure 4. SPST being Pruned With size n=7 entries

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 prune our tree leaves first, having the tree in Figure 4(b) as a result. Also, value 21 is not inserted into the index structure because it is already indexed by the range 20 and 30. 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.

4. Preliminary Results

We performed initial experiments of the SPST performance implementing a column-store prototype, the discussed Data Structures and Cracking Algorithms in Java 8. We run our implementation in a Mac OS X El Capitan (10.11.5) with 8GB of RAM and a i7-4750HQ CPU @ 2.00GHz as CPU. All data structures including the cracker column are in memory.

We repeat the same experimental protocol described in [Idreos et al. 2007b] that is related only to OLAP workloads. All experiments are based on a single column table with 10^7 tuples (Unique integers in $[1, 10^7]$) and some series of 10^4 range queries. There are two different range query groups, one is random selected from the 10^7 integers and the other one is selected by a ZipF distribution [Breslau et al. 1999]. According to ZipF's law the most frequent elements will occur approximately twice as often as the second most frequent element, three times as often as the third most frequent element, and so on.



(a) Random Workload (b) Skewed Workload

Figure 5. Accumulated Query Response Time

Figure 5(a) depicts the accumulated query response time for $10,000$ queries of type $(a < x < b)$, where a and b are chosen randomly from the unique integers in $[1, 10^7]$. Response times were 12% better for SPST than for AVL.

Figure 5(b) depicts the accumulated query response time for $100,000$ queries of type $(a < x < b)$, where for the first $10,000$ queries a and b are chosen randomly from the unique integers in $[1, 10^7]$ and the remaining queries are chosen from a ZipF distribution. We notice that the SPST starts cheaper than the AVL Tree, but the difference becomes more notable when it reaches the mark of $10,000$ queries. This is due to the clusterization of the most accessed nodes near the root of the SPST. Also, at the mark of $15,000$ and $61,000$ queries, we notice small jumps in the slope. These jumps are due to nodes selected outside the most cracked pieces, making cracking operations executed in parts of the cracker column that were not much partitioned. Response times were 20% better for SPST than for AVL.

5. Future Work

Our research agenda to finish the master dissertation, includes assessing the impact of our pruning strategy with the following scenarios: (1) Low frequency high volume updates (2) High frequency low volume updates. With these scenarios we plan to compare the performance of the AVL Tree to our SPST using different pruning configurations. We also intend to show a cost breakdown of the updates algorithms in each data structure to analyze where the engine spends most of the time and how that changes over time. We have considered only the AVL and Splay Tree as Cracker Indices, so we also plan to compare our tree with other sophisticated data structures, such as ART Tree [Leis et al. 2013], that is a main-memory optimized search tree suggested by [Schuhknecht et al. 2013].

Acknowledgments

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

References

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

Breslau, L., Cao, P., Fan, L., Phillips, G., and Shenker, S. (1999). Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings*, pages 126–134.

Elmasri and Navathe (2007). Sistemas de banco de dados. Pearson.

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

Idreos, S., Kersten, M. L., and Manegold, S. (2007a). Updating a cracked database. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 413–424.

Idreos, S., Kersten, M. L., and Manegold, S. (2009). Self-organizing tuple reconstruction in column-stores. pages 297–308. ACM.

Idreos, S., Kersten, M. L., Manegold, S., et al. (2007b). Database cracking. In *CIDR*, pages 1–8.

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

Leis, V., Kemper, A., and Neumann, T. (2013). The adaptive radix tree: Artful indexing for main-memory databases. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 38–49.

Schuhknecht, F. M., Jindal, A., and Dittrich, J. (2013). The uncracked pieces in database cracking. In VLDB, editor, *Proceedings of the VLDB Endowment*, pages 97–108.

Sleator, D. D. and Tarjan, R. E. (1985). Self-adjusting binary search trees. pages 652–686.