Universidade Federal do Paraná

Pedro Thiago Timbó Holanda

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

Curitiba PR
2017

Pedro Thiago Timbó Holanda

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

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Informática no Programa de Pós-Graduação em Informática, setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Prof. Dr. Eduardo Cunha de Almeida.

Curitiba PR
2017

Holanda, Pedro Thiago Timbó
SPST-Index: a self pruning splay tree index for database cracking / Pedro Thiago Timbó Holanda. – Curitiba, 2017
43 f. : il.; tabs., grafs.

Dissertação (mestrado) – Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.
Orientador: Eduardo Cunha de Almeida

1. Banco de dados. 2. Índices. 3. Estruturas de dados (Computação). I. Almeida, Eduardo Cunha de. II. Título

CDD 005.73

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
Setor CIÊNCIAS EXATAS
Programa de Pós-Graduação INFORMÁTICA

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **PEDRO THIAGO TIMBÓ HOLANDA** intitulada: **SPST-Index: A Self-Pruning Splay Tree Index for Caching Database Cracking**, após terem inquirido o aluno e realizado a avaliação do trabalho, são de parecer pela sua aprovação.

Curitiba, 24 de Fevereiro de 2017.

EDUARDO CUNHA DE ALMEIDA
Presidente da Banca Examinadora (UFPR)

MURILO VICENTE GONÇALVES DA SILVA
Avaliador Externo (UTFPR)

CARLOS ALBERTO MAZIERO
Avaliador Interno (UFPR)

"The isolated man does not develop any intellectual power. It is necessary for him to be immersed in an environment of other men, whose techniques he absorbs during the first twenty years of his life. He may then perhaps do a little research of his own and make a very few discoveries which are passed on to other men. From this point of view the search for new techniques must be regarded as carried out by the human community as a whole, rather than by individuals." - Alan Turing.

Agradecimentos

Como agradecimento mais especial, a minha família, a maior patrocinadora de todos os meus sucessos. Meus pais, Tarcisio Holanda e Ana Timbó, e minha irmã, Camila Holanda, pelos diversos incentivos aos meus estudos acadêmicos, por todo o carinho, paciência e amor que deram em toda a minha vida.

Aos meus amigos de laboratório, Walmir Couto, Diego Tomé, Edson Ramiro e Leandro Almeida, por me ajudarem a acreditar na relevância do meu trabalho e pelas incontáveis resenhas nas horas de café.

Aos meus colegas da UFPR, pelas reuniões semanais, sempre tendo críticas extremamente construtivas sobre o meu trabalho.

Por fim, ao meu orientador, professor Dr. Eduardo Cunha de Almeida, cujo auxílio foi além do desenvolvimento desse trabalho. Meu sincero agradecimento por ter compartilhado seu conhecimento diariamente e por ter sido uma parte tão relevante no meu desenvolvimento acadêmico.

Resumo

Em *Database Cracking*, uma coluna de banco de dados se organiza fisicamente, de maneira autônoma, em partições, um índice é então criado para otimizar o acesso a essas partições. A árvore AVL é a estrutura de dados utilizada para implementar esse índice. Contudo, em termos de *cache*, ela é particularmente ineficiente para consultas de intervalos, já que seus nós acessados apenas algumas vezes e os nós frequentemente acessados estão espalhados por toda a árvore. Esse trabalho apresenta a *Self-Pruning Splay Tree (SPST)* que é uma estrutura de dados capaz de reorganizar os dados mais e menos acessados, melhorando o tempo de acesso para as partições mais acessadas. Para cada consulta de intervalo, a SPST rotaciona para a raiz os nós que apontam para os valores do predicado da consulta e o valor médio do intervalo. Eventualmente, os nós mais acessados da árvore irão permanecer próximos a raíz, melhorando a utilização da CPU e a atividade de *cache*. Os nós menos acessados permanecerão próximos às folhas e serão removidos para limparmos dados que não são utilizados, diminuindo o tamanho do índice e obtendo custos de leitura e atualização menores.

Palavras-chave: *Database Cracking*, Índice para *Cracking* , Árvore *Splay*.

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 current 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. This work presents 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 clean up unused data in order to diminish the storage footprint with significant improvements: smaller lookup/update costs.

Keywords: Database Cracking, Cracker Index, Splay Tree.

Contents

List of Figures

List of Tables

Lista de Acrônimos

ART	Adaptative Radix Tree
AVL	Adelson-Velsky Landis
BST	Binary Search Tree
CPU	Central Processing Unit
CSSL	Cache Sensitive Skip List
DBA	Database Administrator
DBI	Disk-based Indexes
DBMS	Database Management System
DINF	Departamento de Informática
DSM	Decomposition Storage Model
FCRT	Fractional Cascading Range Tree
HBT	Height Balanced Trees
HFLV	High Frequency Low Volume
IPC	Instructions Per Cycle
ITT	Indexing Total Time
LFHV	Low Frequency High Volume
MCI	Merge Complete Insertions
MGI	Merge Gradually Insertions
MRI	Merge Riple Insertions
NP	Non-deterministic Polynomial time
NSM	N-ary Storage Model
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
PPGINF	Programa de Pós-Graduação em Informática
PT	Pruning Time
QTT	Query Total Time
SPST	Self Pruning Splay Tree
UFPR	Universidade Federal do Paraná
UTT	Update Total Time

Chapter 1

Introduction

Indices are data access methods. They 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 [15].

However, creating indexes is not a simple task. Knowing which indexes 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) and storage quota. 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).

The Database Management System (DBMS) must be able to internally find the best physical design to react to its own workload since it uses a declarative language [8].

A Columnar DBMS [1] vertically partitions a database in collections of columns that can be stored separately, called column-oriented databases. Because of the partitions, queries only read the attributes they need, instead of reading the full tuple and later discarding the unnecessary attributes fetched from disk to memory (i.e., Row DBMS [15]). Thus, if the query needs only a small number of columns of many tuples, the number of searches is inferior of a Row DBMS. That way, Columnar DBMS is a better fit than Row DBMS for Online Analytical Processing (OLAP) workloads.

OLAP workloads are very dynamic, changing their access pattern overtime, making the Self-Tuning Tools job harder, since its approach depends on predicting future workloads.

Database Cracking [25] has been proposed for column-oriented 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 $A > V_1$, where $V_1 \geq 10$, 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 [25].

Another benefit of database cracking is the access method to reach the created pieces. Typically, an index is a data access method that stores a list of pointers to all disk blocks. A drawback of the current index structures is the storage footprint that sometimes is as large as

the column size [27]. 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 partition to boost access to an interval of values. The advantage is the direct access to the beginning of each partition. Besides, the index storage footprint is smaller than regular indices.

The current data structure implemented as a cracker index is the self-balancing AVL Tree [5], 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, it 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 and the storage footprint growing to the size of the column size itself.

A Splay Tree [32] is a self-adjusting binary search tree (BST) 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 incoming workload, clustering "Hot Data" at the root of the tree. Therefore, the most frequent accessed nodes will be accessed faster. The rarely accessed nodes are stored closer to the leaves, giving the opportunity to prune cold data.

Our index structure, called Self-Pruning Splay Tree (SPST) aims to identify and keep "Hot Data" close to the root of the tree [21]. The SPST explores the *Splaying* operations to rotate the values accessed by range queries. In particular, the SPST 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. In this work, we present a pruning strategy to keep a small storage footprint for our index while keeping the lookup performance for "Hot Data". "Cold Data" is stored close to the leaves presenting the opportunity to prune them out of the index. We present our assumptions and strategies to prune the tree values that eventually improve maintenance and shrink the storage footprint without any compromises to access "Hot Data". 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 1% even in the worst case scenario with mixed workloads (i.e., lookups and batch updates) while having 25% of the AVL size.

This dissertation aims to contribute with the following:

- A discussion of different data structures for indexing range lookups;
- the SPST, a data structure to cache "Hot Data" and prune "Cold Data";
- the algorithms to build and maintain the SPST in order to optimize CPU caching and concurrent access to cracked pieces;
- a discussion of CPU and caching improvements of the SPST compared to the AVL across different data access patterns.
- an analysis of the cost breakdown of the AVL converging to full-index, while the SPST remain small upon mixed workloads with low frequency/high volume updates (LFHV) and high frequency/low volume updates (HFLV);

- a discussion of pros/cons of the K-Splaying optimization to reduce the number of rotations of the SPST;
- a thorough experimentation of the pruning strategy to shrink the storage footprint and reduce lookup and update costs;

The remainder of this dissertation is organized as follows. We introduce Database Cracking in Chapter 2. We present the related work in Chapter 3. We describe our solution in Chapter 4. The analysis and corresponding results are presented in Chapter 5. Finally, we conclude the dissertation and present future work in Chapter 6.

Chapter 2

Database Cracking

In this dissertation we focus on Database Cracking for Column Oriented DBMS. We briefly describe the Row Oriented DBMS in Section 2.1. The Column Oriented DBMS is presented in Section 2.2. In section 2.3 we compare both Row and Column Oriented DBMS. In Section 2.4 we describe different Self-Tuning Approaches. Finally, we give an overview on Database Cracking in section 2.5.

2.1 Row Oriented DBMS

The way data is stored in disk defines how the DBMS is able to process its queries [15]. Traditional Row Oriented DBMS sorts and processes data in a tuple format, one tuple at a time.

The storage is usually done in pages, with 4k of data, and a certain number of rows from a table. Since we have many different possibilities of data types in a table, it is necessary to maintain metadata for each page, row and row attribute, data size, the position which rows and attributes start, and other necessary information to go through a page.

When a query is being processed, it must navigate through pages and find the requested attributes, job that needs to be executed across an entire table.

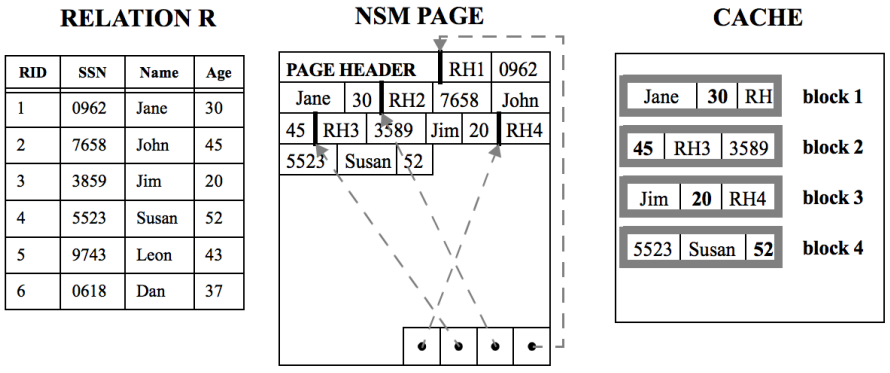


Figure 2.1: The N-ary Storage Model (NSM) and its cache behavior. [3]

Figure 2.1 depicts an example of storage in a Row DBMS. Records in R (left) are stored contiguously into disk pages (middle). Supposing a query that only scans age, NSM brings useless data into the cache (right).

2.2 Column Oriented DBMS

The Decomposition Storage Model (DSM) vertically partitions an n-attribute relation into columns, each of which is accessed only when queries need it [13]. A Column Oriented DBMS uses the DSM and vertically partitions a database into a collection of columns that are stored separately. Since the columns are separated, queries are able to read just the needed attributes to answer a predicate, instead of reading the full tuple and later on selecting the interesting attributes.

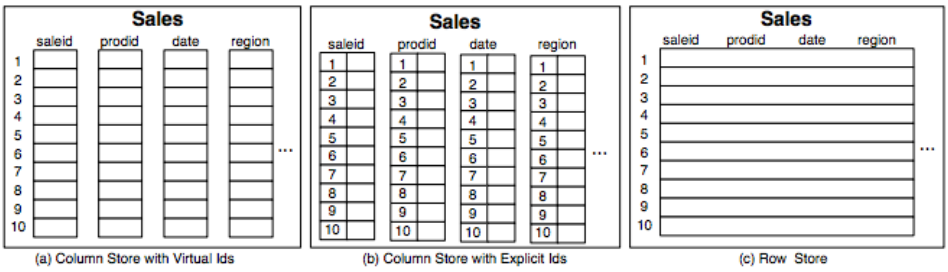


Figure 2.2: Physical layout of column and row oriented databases. [1]

Figure 2.2 depicts an example of three different ways to store a table that contains various attributes, in order to show the differences in physical layout of Column and Row Oriented DBMS. Figure 2.2 (a) and Figure 2.2 (b) depicts the two possible ways in Column Oriented DBMS where each column from the table is stored separately, meaning that each block holds data for one of the columns only. If a query processes only the number of sales of a particular product in a particular month, it only needs to access the "prodid" and "date" columns from disk to memory. The difference between those two ways of storing data is that in Figure 2.2 (a) we have virtual "ids", which mean we reconstruct the tuple by the id of its column position, so tuple 1 must be "saleid(1)","prodid(1)","date(1)" and "region(1)". But in Figure 2.2 (b) when we have explicit ids this order does not have to be maintained so tuple 1 could be in any position of its attributes. Figure 2.2 (c) depicts the Row Oriented approach, where the columns are stored as rows together, with multiple columns being stored in the same block and following the row order.

2.3 Row vs Column Oriented DBMS

There are several differences between Row and Column Oriented DBMS, our main interest is in the access patterns regarding the workload, which tell if a column or row oriented has a better physical layout. If a query selects one or more records (i.e., one or more columns from a table), a column oriented DBMS needs to seek several columns to read one record entirely, but if only a small number of columns needs to be brought from disk, then large partitions of entire columns can be read, amortizing the seeks to different columns, this is particularly convenient for OLAP. In a Row Oriented DMBS is the other way around, if a query selects one or more records it will do one seek per record. But if selects a small number of columns from the entire table the transfer time begins to be larger than the seek time. [1]

2.4 Database Indexing

An index is a data structure that contains all data for some selected attributes and its main objective is to store the data in a different way compared to the table, to boost performance of data access.

Nowadays, we have four approaches regarding indexing to query processing.

Figure 2.3: Four approaches to indexing with regard to query processing. [18]

Figure 2.3 depicts those approaches, in yellow we have the time spent to perform a full scan, in red the index creation time and in green lookup times while accessing the created index. The approaches are:

- The data is loaded without indexes and then for every query a table scan is done (See Figure 2.3 (1)).
- The offline approach of self-tuning tools where is done an investment in order to create certain indexes. Later on, the queries can use those indexed columns executing faster (See Figure 2.3 (2)).
- The online approach of current self-tuning tools where the data is initially loaded without indexes and the tool takes some time to analyze the incoming workload, being able to construct the most promising indexes (See Figure 2.3 (3)).
- The Database Cracking, which is an adaptive indexing technique, creating and refining indexes incrementally as a side product of query execution(See Figure 2.3 (4)).

Following [8] we know that many self-tuning tools have been developed to monitor a DBMS, by selecting relevant workloads, identifying and creating indexes. This is done by collecting statistics and using the What-if architecture [9]. The What-if architecture allows the tools to create hypothetic indexes, estimating its creation costs and then, making the optimizer simulate the behavior of the workload assuming those indexes are available. The search for indexes is a NP-Hard problem [11], nowadays those tools are able to use many different techniques to analyze good plans and later suggest indexes for the DBA to create.

The Self-tuning tools approaches are ideal for scenarios where:

- We have workload knowledge, with a workload that is very unlikely to change
- We have much idle time and space to invest in index creation/maintenance

For dynamic and unpredictable workloads, adaptive indexing is a better suit since there is no workload knowledge and no time to spend creating indexes.

2.5 Database Cracking

Database Cracking is created as an index maintenance approach that creates indexes as a side product of query processing. Each incoming query starts to be interpreted as a hint on how to crack the physical database design into small manageable pieces. Each of those pieces is referenced by a cracker index, which substitutes the common indexes.

It was inspired by Column Oriented DBMS because of its flexibility to manipulate and reorganize the columns independently. Since the columns are stored separately, it is possible to organize one column without affecting the remaining columns. Database Cracking works as follows:

- The first time a range query selects an attribute A, it creates a copy of column A. Called Cracker Column of A, A_{crk}
- The Cracker Column A_{crk} is continuously reorganized every time a query selects its attribute.

The cracker column is physically reorganized incrementally based on queries that are executed. At the same time an index is created to keep track of the column pieces. Cracking becomes particularly interesting in scenarios where:

- There is no previous knowledge about the workload.
- The workload changes constantly (i.e., OLAP).
- There is no knowledge regarding the most accessed attributes from a table.
- There is no idle time available.
- It is hard to keep many indexes due to frequent updates.

2.5.1 Read Operations

There are two Database Cracking algorithms to split the columns into two and three partitions, called *crack-in-two* and *crack-in-three* 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. The performance of database cracking starts similar to a full column scan and overtime gets close to the performance of a full index.

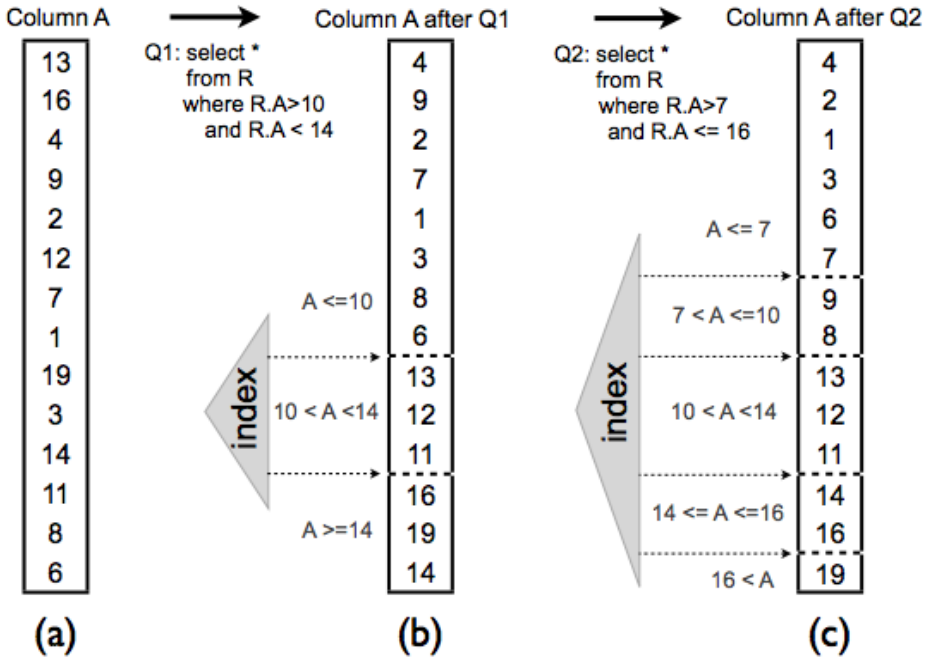


Figure 2.4: Database Cracking when executing two queries with different ranges from [31]

Figure 2.4 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.

2.5.2 Cracker Index

The cracker column is being continuously partitioned into more and more logical pieces as queries are being processed. It is necessary another data structure, called cracker index, to assemble the information regarding the pieces from the cracker column.

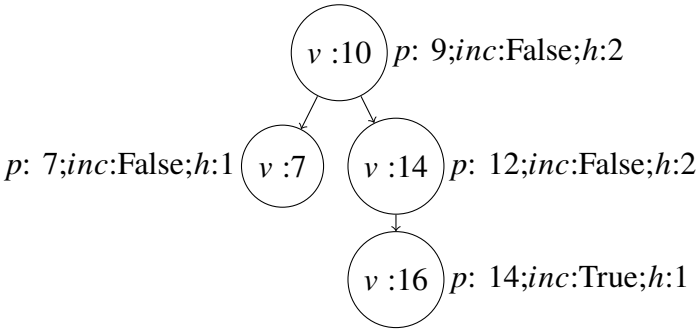


Figure 2.5: Cracker Index (AVL Tree) from Figure 2.4

The current data structure used as a Cracker Index [25] is an AVL Tree which is a BST (i.e., a tree with a finite set of nodes where all keys in the left subtree of a node are have smaller keys compared to the parent key and all the trees in the right subtree have greater keys compared to the parent) such that for every non leaf node, the height of the left and right subtrees differ by at most one. This is maintained by a field in every node that maintains the height of the right subtree minus the left subtree. To keep itself balanced it performs a combination of rotations. Lookup, Insertion and Deletion take $\Theta(logn)$ time in average and worst cases, where n is the number of nodes in the tree. Insertions and Deletions may require the tree to be rebalanced by one or more tree rotations [5]. Figure 2.5 depicts an AVL Tree as a Cracker Index to the Cracker Column in Figure 2.4.

As a Cracker Index, each node of the AVL Tree stores:

- The indexed range predicate v.
- The storing position p that refers to the column partition.
- The variable inc to check if v in included in p.
- The subtree height h.

2.5.3 Database Cracking Algorithms

The discussed algorithm is known as the standard database cracking. However, there are other cracking approaches in the literature, including:

- *Hybrid Cracking* is created to address the issue of poor convergence of standard cracking into full index [26]. Initially, it creates unsorted initial pieces that are physically reorganized and then adaptively merged for faster convergence to a full index.
- *Sideways Cracking* is created to address the issue of inefficient tuple reconstruction in standard cracking [24]. It minimizes the tuple reconstruction cost by using a data structure called cracker maps. They provide a mapping between attributes that are combined in queries.
- *Stochastic Cracking* is created to address the issue of performance unpredictability in database cracking [19]. It creates partitions using a random pivot element. The pivot is used to perform arbitrary reorganization steps for more robust query performance.

2.5.4 Write Operations

To update a cracked database there are two basic structures to consider: the cracker column and the cracker index. The cracker index I maintains information about the cracked pieces of the 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.

The main idea is to handle any updates without losing information from the cracker index and doing it only when needed. An update operation starts at the moment an incoming query requests any of the ranges that have values already inside the pending insertion column.

Figure 2.6 depicts an example of insertion of the value 17. Let us assume an incoming query with range $5 < A < 50$. The value 17 is already part of the pending insertions column and also part of the requested range. In Figure 2.6 (b) the value 17 is placed in the second cracked

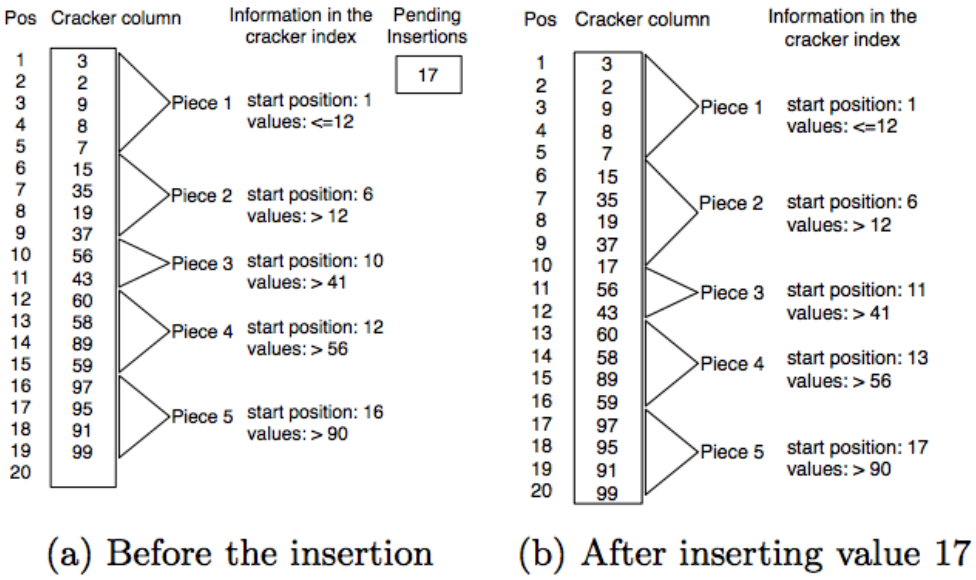


Figure 2.6: A write operation in a cracked column using the pending insertions column [23]

piece, because this partition holds the tuples within the range $12 < v \leq 41$. After inserting a new element in the cracked column, the cracker index is also updated along with the information regarding the start position of pieces 3, 4 and 5.

The *Shuffle Algorithm* is in charge of merging a sorted portion of the pending insertions column into the cracker column. In each piece p_i, the algorithm first inserts any values from the the pending insertions column that are in the range of p_i. The remaining tuples are moved from the beginning of p_i to its end. The algorithm continues this merging process as long as there are pending insertions.

Three different merging algorithms based on the *Shuffle Algorithm* are presented in [23]. They are:

- Merge Complete Insertions (MCI). If a query requests any value from the pending insertions column, all insertions are placed in the cracker column;
- Merge Gradually Insertions (MGI). If a range query needs only k tuples from the Pending Insertions Column, it inserts only these k tuples into the cracker column;
- Merge Ripple Insertions (MRI). Uses the same strategy from MCI but avoiding unnecessary shuffles. This is achieved by adding, to the pending insertions columns, unnecessary values for a running query and removing them from the cracker column, that way creating space for the necessary insertions.

The MRI works by making enough space for the merging process. To merge k tuples, the MRI starts directly at the position after the last tuple of piece p_h, where the tuple with the highest qualifying value belongs to. From there, k tuples are moved into a temporary buffer. Then, the *Shuffle Algorithm* runs for the qualifying portion of the pending insertions column. MRI differs from MGI or MCI by merging from piece p_h and not from the last piece of the cracker column. Finally, the tuples in the buffer are merged into the pending insertions column. Merging these tuples back in the cracker column is left for future queries.

Figure 2.6 shows an example using the MRI algorithm. Piece 3 contains the highest qualifying value for the incoming range request $5 < A < 50$. With this incoming query, the value 17 is going to be merged. The value 60 is moved from position 12 in the cracker column to the buffer. Then the *Shuffle Algorithm* starts making space in Piece 3. The value 17 does not belong in Piece 3 so the value 56 (first value in Piece 3) is moved from position 10 to position 11 opening space to place the value 17 and updating the start position of Piece 3. The cracker index is also updated so that Pieces 3 and 4 have their starting positions increased by one. Finally, the tuple with value 17 is moved from the buffer to the pending insertions. At this point MRI finishes without having shifted Pieces 4 and 5 as the other merge algorithms would have done.

Chapter 3

Related Work

In this chapter, we discuss the complexity of different data structures for indexing range scans. However, not all of them are implemented as Cracker Indexes (see Table 3.1). Also, not all data structures are Cache Conscious. We discuss the SPST range scan and space complexity in the next chapter, after presenting the SPST.

Table 3.1: Comparison of Data Structures for Indexing Range Lookups.

Data Structure	Range Scan (Average Case)	Space Complexity	Cache Conscious	Cracker Index
AVL [5]	$k + \log n$	$\|P\| \rightarrow n$		X
RedBlack [20]	$k + \log n$	n		
2-3 Trees [34]	$k + \log n$	n		
B+Tree [12, 6]	$k + \log_B n$	n		
BlockRange [35]	$k + \log n$	n		
FCRT [10]	$k + \log n^{d-1}$	$n \log n^{d-1}$		
Art [29]	$k + n$	$n + 1 \rightarrow a + n$	X	
CSSL [33]	$k + \log n$	$n \log n$	X	

3.1 Height Balanced Trees

The AVL, RedBlack and 2-3 Trees are height balanced trees with different self-balancing properties. The AVL uses a balance-factor to ensure that if in a given time the height of two child subtrees of any node is bigger than one, a series of rotations are triggered in order to rebalance the tree.

In the RedBlack Tree each node of the binary tree is colored to keep the height property. A red node must have a black parent and the number of black nodes must be the same to every branch. The number of black nodes is the height of the tree. In the 2-3 Tree every node has 1 or 2 keys and all the leaves are stored in the same level of the tree. Its nodes can be split or merged to maintain the tree balanced.

In database cracking, any height balanced trees draw the same result spreading both Hot and Cold data across the entire index. This leads to more pointer chasing with poor CPU and cache activity. The complexity of performing a range scan in all height balanced trees is the same: $\Theta(k + \log n)$, where k is the number of points in the range and n is the size of the tree. The Space Complexity is $O(n)$. However, with the AVL Tree cracker index implementation the

space complexity depends on the number of $O(|P|)$ partitions, although $|P|$ converges to full index n as more queries are executed.

3.2 Disk-based Indexes

The B+Tree and the BlockRange are data structures designed for disk-based database systems, where the main objective is to avoid unnecessary disk I/Os (i.e., read and write operations). In the B+Tree each internal node stores keys and child pointers, but the record pointers are stored on leaf nodes only. Multiple keys are used to search within a node and they have a high fanout (i.e., number of pointers to child nodes in a node). The complexity of performing a lookup is $\Theta(k + \log_B n)$, where B is the block size. There are also improvements on the B+Tree for boosting traversal with $\Theta(\log_B n + \log_\alpha \log_{10} B)$, where α is a parametrized constant to affect the height of the tree [6].

The BlockRange keeps track of min and max values in consecutive page ranges and works similar to a SkipList, but with a range of keys in each block instead of maintaining multiple pointers to each part of the list. In the SkipList, the worst case goes to $N + h$, where h is the number of stages (or SkipLists), but in the case of the BlockRange there is only one SkipList and $h = 1$. Therefore, in the average case the BlockRange goes to $\Theta(k + \log n)$. The Space Complexity of these data structures is $O(n)$ for n indexed registers. Anyhow, all of these disk-based indexes are not designed to take advantage of the most frequently accessed data prioritizing its access in cache.

3.3 Other Data Structures

The Fractional Cascading in Range Trees (FCRT) is a technique used to improve the range scans for the same value. It starts with logarithmic complexity, but successive searches become faster. Becoming able to achieve a complexity of $\Theta(\log n^{d-1} + k)$ when applied to the KD-Trees [7], where k is the number of points in the range, n is the size of the tree and d the number of dimensions. When applied to binary search trees, it maintains the complexity of $\Theta(k + \log n)$. The FCRT have a space complexity of $O(n \log n^{d-1})$, where $\log n^{d-1}$ accounts for the stored dimensions.

The Adaptive Radix Tree (ART) is a recent radix tree variant designed for main memory to address the shortcomings of balanced trees in modern CPU/RAM. Its main goal is to be space efficient by adaptively choosing compact and efficient data structures for internal nodes. The ART uses different compression techniques to ensure that data always fit in main memory. The complexity of performing a range scan is $\Theta(k + n)$ time, where n is the length of the key and k the number of points in the range. The ART have a space complexity of $O(a + n)$ as its worst case, where n is the number of keys and a is the alphabet and $\Omega(n + 1)$ as its best case when the alphabet equals to 1.

The Cache-Sensitive Skip List (CSSL) is an alternative implementation for balanced skip lists and height balanced trees. Its main goal is to keep fast lanes as separate entities in dense arrays, allowing the use of SIMD instructions. The CSSL have range scan complexity of $\Theta(k + \log n)$ where k is the number of points in the range and n is the number of the elements in the CSSL. The CSSL is able to exploit modern CPUs by reducing the number of pointers in the list. The CSSL have space complexity of $O(n \log n)$, where n is the number of keys stored in the list and $\log n$ is the keys stored in the fast lanes.

Chapter 4

The SPST-Index

In this chapter we discuss the main concepts of the SPST and a thorough description of the algorithms to build and maintain the SPST to leverage CPU caching and optimize the access to cracked pieces, also to reduce the number of rotations to diminish concurrency contention on the data structure. With Hot Data cached near the root of the SPST, we expect less cache misses, higher Instructions Per Cycle (IPC) and improvements in the response time of lookups and updates. The SPST is inspired on Splay Trees but with focus on recognizing hot data and pruning cold data for interval queries. It is developed to work in compliance with database cracking as a cracker index. Its main benefits are: Improving access time to data, improving CPU and cache activity, diminishing the index storage footprint and boosting updates. The construction of the SPST is divided in two parts: the algorithm to rotate the pointers to the range of the query intervals and the strategy to diminish the number of rotations expected in Splay trees. We present in Section 4.1 the definition of Splay Trees. Section 4.2 presents the Range Rotation operation. In Section 4.3 we present the Pruning Operation. In Section 4.4 we present the SPST's Range Scan and Space Complexity. Finally, we present a brief discussion regarding Concurrency Control in Section 4.5.

4.1 Splay Tree

A Splay Tree [32] is a self-adjusting BST 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. Assume a set of elements S, to be stored at the nodes of the tree. Since our data structure is an index, the indexed keys do not repeat. Splay trees keep the invariant of BSTs that is to every node u in tree T the keys of the left subtree of u are smaller than the key in u and all the keys in the right subtree of u are greater than the key in u. To access an element $x \in S$ we start at the root of T and at every node we go to the left or right node to keep the BST invariant. Every time an element x is accessed the node that holds u is moved to the root of the tree performing a sequence of rotations.

4.1.1 Splay Tree Rotations

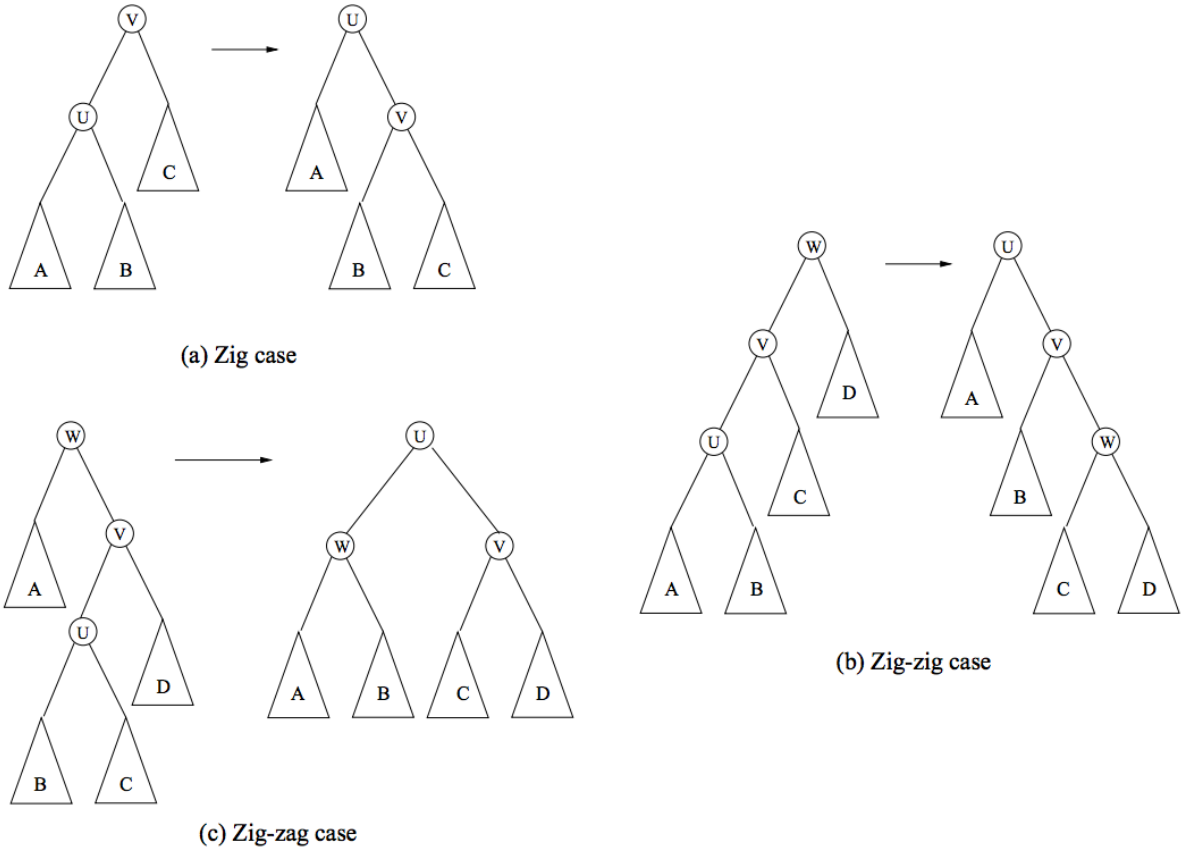
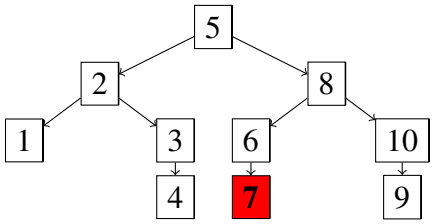


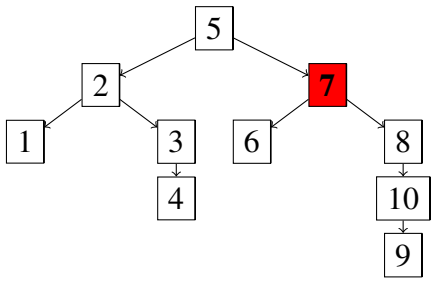
Figure 4.1: Cases of splaying [16]

Figure 4.1 depicts the possible rotations that a Splay Tree can perform in order to bring a node to the root. Where u is the splayed node.

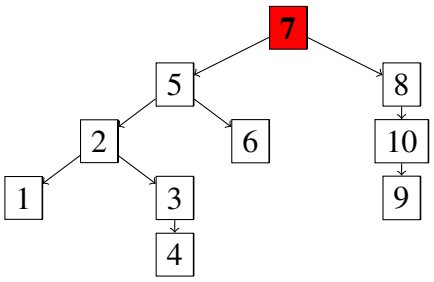
- Figure 4.1(a) depicts a simple Zig rotation. The father of u, $p(u)$ is the root of T. We rotate the edge between u and $p(u)$.
- Figure 4.1(b) depicts a double Zig-Zig rotation. $p(u)$ is not the root and u and $p(u)$ are both left children. First we rotate the edge between $p(u)$ and its grandfather $g(u)$. Then we rotate the edge between u and $p(u)$.
- Figure 4.1(c) depicts a double Zig-Zag rotation. $p(u)$ is not the root, u is left child of $p(u)$ and $p(u)$ is a right child of W. First we rotate the edge between u and $p(u)$. Then we rotate the edge between u and new $p(u)$.



(a) Original Splay Tree



(b) Splay Tree after Zig-Zag rotation on node with key 7



(c) Splay Tree after Zig rotation on node with key 7

Figure 4.2: Splay Tree performing a lookup on key 7

Figure 4.2 depicts an example where a lookup is performed on key 7. After finding key 7, using a standard BST lookup, the Splay Tree depicted in Figure 4.2(a) splays the node with value 7. It starts by performing a Zig-Zag rotation, since the father of 7, is not the root and 7 is a right child with its parent being a left child (i.e., Analogous case to Figure 4.1(c)), resulting in the tree depicted in Figure 4.2(b). It finishes with a Zig rotation, since the father of 7 is the root (i.e., Analogous case to Figure 4.1(a)) resulting in the splay tree depicted in Figure 4.2(c), where the root is the node with key 7.

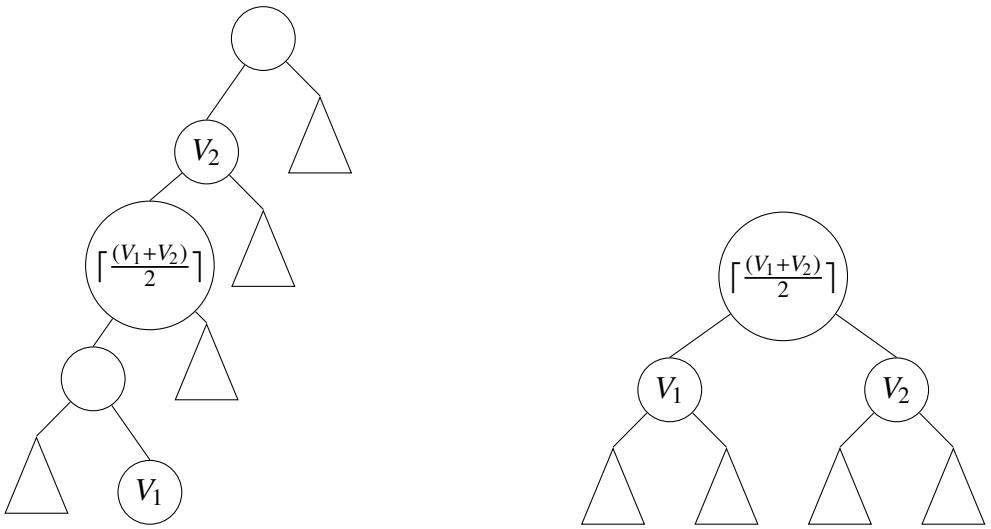
4.2 Range rotation

The SPST was designed to deal with range queries in OLAP. 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 the closest node to the middle.

Listing 4.1: Interval Splay Code Snippet

```
1 SplayTree
2 IntervalSplay( ElementType V1, ElementType V2,
3       SplayTree T )
4 {
5   T = Splay(V1, T);
6   T = Splay(V2, T);
7   T = Splay((V1 + V2)/2, T);
8   return T;
9 }
```

Figure 4.3 depicts a generic interval splay, where the values of V_1, V_2 and $\lceil \frac{(V_1+V_2)}{2} \rceil$ are spread over the tree as depicted in Figure 4.3(a). When a range query of $V_1 < A < V_2$ is executed,

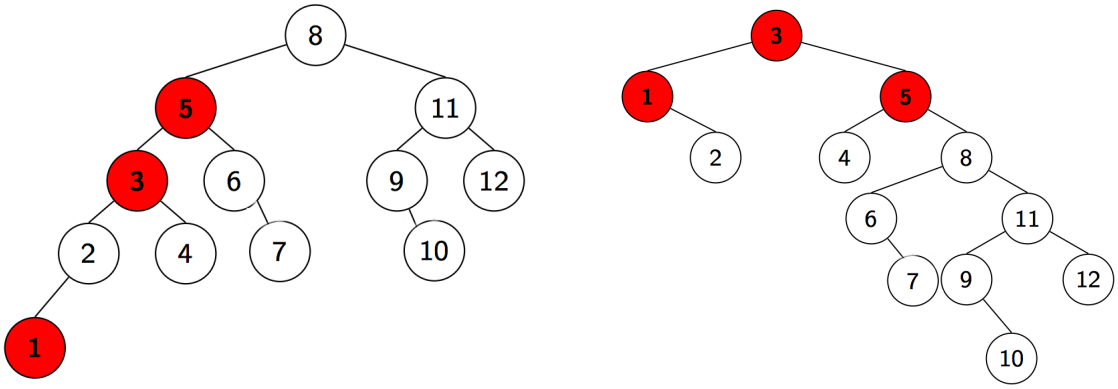


(a) Original SPST

(b) SPST after Interval Splay of V_1 and V_2

Figure 4.3: Interval Splay

the following splay operations are made. Splay (V_1), Splay (V_2), Splay ($\lceil \frac{(V_1+V_2)}{2} \rceil$). Figure 4.3(b) depicts the result, with the interval being stored close to the root, resulting into fast access to the same interval in the future. Listing 4.1 depicts the Interval Splay method that receive both values V_1, V_2 and the Splay Tree T, returning the Splay Tree T with the correct nodes clustered at the root as result.



(a) Original Tree

(b) Tree after Splaying nodes 1, 5, and 3

Figure 4.4: The Splay Tree index with query $1 < A < 5$

Let us consider for cracker index the Splay Tree depicted by Figure 4.4. 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 4.4(b) depicts the resulting tree with nodes 1, 3 and 5 close to the root. In Splay trees, 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.

4.2.1 Reducing Rotations with K-Splaying

As the SPST grows with incoming queries, depending of the skewness of the workload it may present more number of rotations than the AVL Tree. A rotation requires a number of pointer operations executed in CPU. The main problem in tree structures is to traverse root to leaf and the operations to rotate a node to the root may consume CPU cycles. In the SPST the problem of tree traversal is mitigated over time with "Hot data" remaining way up on the tree. However, every lookup still consumes rotations and CPU cycles.

To reduce the number of rotations, we apply the K-Splaying technique proposed in [28]. The K-Splaying technique consists of performing $\frac{1}{k}$ standard lookups and $1 - \frac{1}{k}$ splays. The choice of k is arbitrary but in [4] authors suggest that $k = 2$ (i.e., alternating between splaying and standard lookup) improves the total time in almost all cases. Lower k values, means that less nodes will be splayed, reducing the total amount of rotations but becoming less able to recognize hot data. Our goal with K-Splaying is to see if there is a k value that shows a gain in response time while reducing the number of rotations. Listing 4.2 depicts the K-Splaying code, returning true when the node is splayed and false when it is not.

Listing 4.2: K-Splaying Code Snippet

```
bool
KSplaying(int k) {
      int aux = rand() % k + 1
      if (aux < k)
            return false;
   return true;
}
```

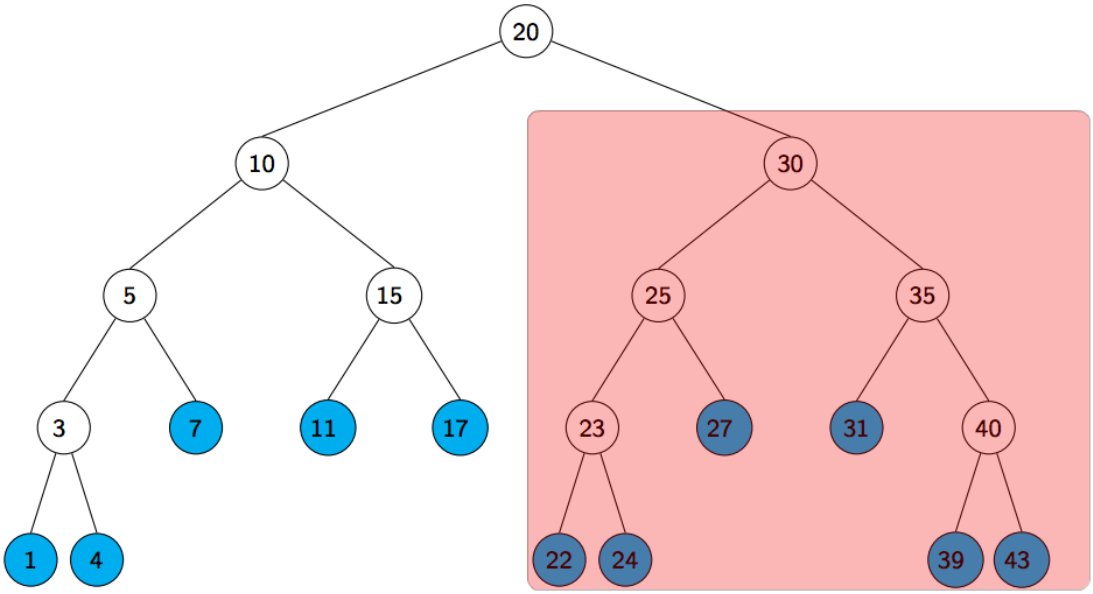
4.3 Pruning

Besides speeding up the access to "Hot data", another goal of the SPST is to speed up updates and maintenance costs. We assume that eventually the nodes stored at the leaves point to "Cold data" opening an opportunity to prune the leaves as a maintenance strategy of our data structure. As we prune them, the expected benefit is decreasing the index size, administration and update costs. 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 Splay Tree.

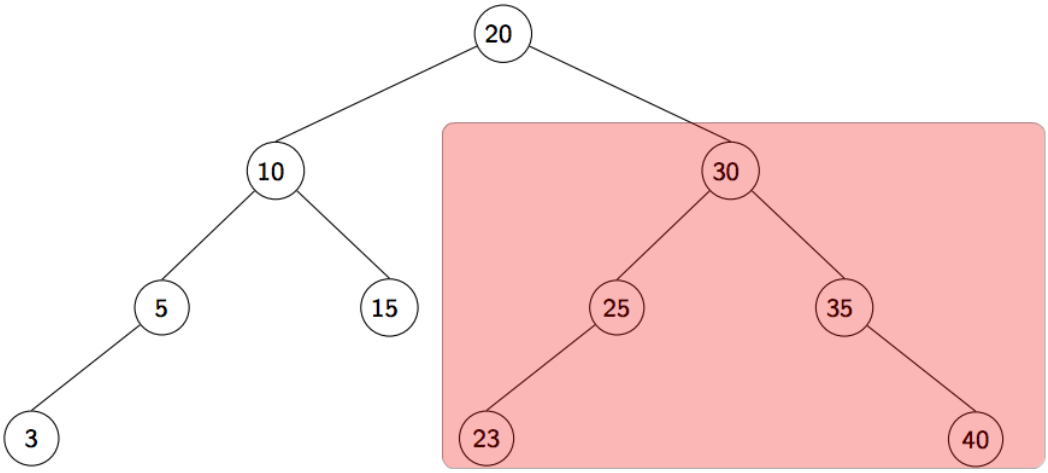
Listing 4.3 depicts the Pruning method that receive the Splay Tree T, remove all of its leaves, and returns the Splay Tree T without the leaves as result.

Listing 4.3: Pruning Code Snippet

```
SplayTree
Pruning(SplayTree T){
   if (T == NULL)
      return NULL;
   if (T->Left == NULL && T->Right == NULL)
      return NULL;
   T->Left = Pruning(T->Left);
   T->Right = Pruning(T->Right);
   return T;
}
```



(a) SPST before Pruning



(b) SPST after Pruning 10 nodes

Figure 4.5: The pruning strategy of the SPST with $\lfloor \frac{|P|}{2} \rfloor$ less nodes as result.

Figure 4.5(a) depicts a generic interval splay where "Cold data" is marked in blue. Before the updates all the "Cold data" is removed from the splay tree, reducing the index size and boosting the update cost. Figure 4.5(b) depicts the resulting tree with 10 nodes out of 21 remaining in the SPST.

Let us suppose the SPST depicted by Figure 4.5(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 update an ordinary Splay tree, we need to update the whole branch of node 22 to splay 21 to the root and also the respective pointers to the cracker column. Instead, in the SPST we prune the leaves having as result the tree depicted by Figure 4.5(b) without the value 21 as it is already indexed by node 20.

4.3.1 Pruning with Standard Deviation

In mixed workloads where the updates occur frequently and in small numbers (e.g., the frequency of one query, one update), pruning before every update may not be beneficial. This can occur since the index can become very small (i.e., far from converge to a full index) making the cracking cost to be extremely high, shadowing the update benefits and resulting in a total cost that is higher than the tree that is not pruned. For that scenario, we evolved our previous pruning strategy by pruning only when the cracking time stabilizes. In this strategy, we calculate the standard deviation of the cracking time for previously executed update queries, as follows:

$$s = \sqrt{\frac{1}{P-1}\sum_{i=1}^{P}(x_i - \overline{x})^2}$$

where x is the set of cracking time values, P is the number of elements in set x and x_i are individual i.

Before an update, the function $pruning(s, d)$ decides if pruning of the SPST is needed. If the standard deviation s is less or equal to a threshold d defined by the user, then it prunes the tree and executes the update, as follows:

$$pruning(s, d) = \begin{cases} \text{True} & \text{if } s \leq d \\ \text{False} & \text{otherwise} \end{cases}$$

4.4 SPST Range Scan and Space Complexity

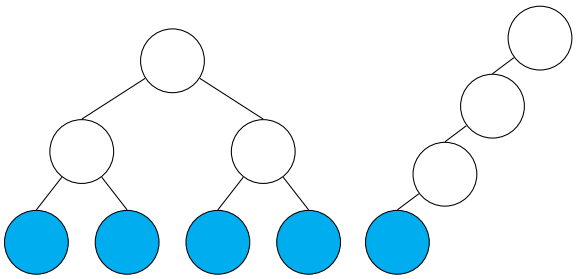
Table 4.1: Comparison of Data Structures for Indexing Range Lookups (including the SPST).

Data Structure	Range Scan (Average Case)	Space Complexity	Cache Conscious	Cracker Index
AVL [5]	$k + \log n$	$\|P\| \rightarrow n$		X
RedBlack [20]	$k + \log n$	n		
2-3 Trees [34]	$k + \log n$	n		
B+Tree [12, 6]	$k + \log_B n$	n		
BlockRange [35]	$k + \log n$	n		
FCRT [10]	$k + \log n^{d-1}$	$n \log n^{d-1}$		
Art [29]	$k + n$	$n + 1 \rightarrow a + n$	X	
CSSL [33]	$k + \log n$	$n \log n$	X	
SPST [21]	$k + \log \|P\|$	$\log(\|P\|) \rightarrow \|P\| - 1$	X	X

The complexity of performing a range scan in the SPST is the same for the AVL Tree (See Table 4.1). The range scan take $\Theta(\log |P|)$ time in the average, where $|P|$ is the number of nodes in the SPST (i.e., number of indexed partitions).

The SPST worst case space complexity is $O(|P| - 1)$ pointers to cracked pieces (See Table 4.1). $|P| - 1$ means the SPST becomes a linked list and there is only one node as leaf to be pruned (see Figure 4.6(b)). This case only happens after an empty tree receives sequential inserts, but the dynamism of the SPST turns the linked list into a tree shape with a few lookups (i.e., a range lookup demands 3 splays in the SPST). The SPST best case space complexity is

$\Omega(\log |P|)$ pointers to cracked pieces if we assume that these pointers are mostly Hot Data and the SPST is balanced and frequently pruned (See Figure 4.6(a)).



(a) Best case scenario (b) Worst case scenario

Figure 4.6: The SPST best/worst case scenarios for space complexity.

4.5 Concurrency Control

The adaptive indexing characteristics of database cracking poses some questions about concurrency control, because read-only queries require structural changes in the index to track the evolving partitioning. A thorough study showing structural modifications to the physical representation of cracking indexes is presented in [17]. This study demonstrates minimal performance overhead of the concurrency control during structural updates, because the logical contents of the index remain unmodified. It also discusses possible techniques to minimize index contention, such as: incremental locking and adaptive early termination.

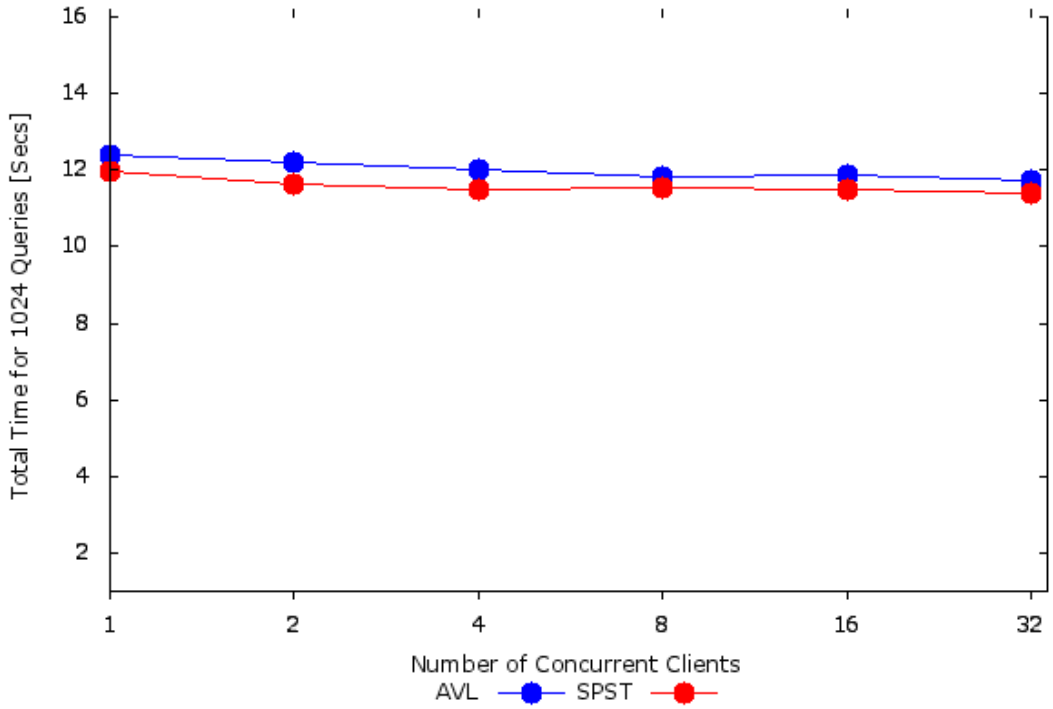


Figure 4.7: Effect of concurrency with 1024 queries and 32 clients

Considering our SPST index, every read-only query brings more updates to the index structure than the AVL tree due to the splay operations. However, there are many possible optimizations to diminish the number of rotations in splaying, including, similarity in working sets [2] and different types of heuristics, like the "K-Splaying" [28] used by the SPST. Therefore, the SPST presents similar performance of concurrency control as the AVL Tree (see Figure 4.7).

We executed the same experimental protocol presented in [17]. The experiment consists of running 1024 random queries, with 10^{-2} selectivity and varying the number of concurrent clients ranging from 1 to 32. The protocol works as follows: one client fires all the 1024 queries, then two clients fire 512 queries each, then 4 clients fire 256 queries each and so on. The experiment uses a selectivity of 10^{-2}, meaning that the queries access a small part of the column and concurrent queries should access different parts of the column.

Chapter 5

Experiments

In this chapter, we report on the experimental evaluation of the SPST implemented as a cracker index. In Section 5.1 we report the impact of read operations and the maintenance of the SPST running the same lookup scenarios described in [31]. In Section 5.2 we report the results of the K-Splaying optimization to reduce the number of rotations. Finally, in Section 5.3 we report the impact of write operations and our pruning strategy by running the update scenarios described in [23].

Overall, we capture four different time measurements, as follows: (i) "Query Total Time" (QTT) is the total time to execute the query after cracking. For this measurement, we capture the time to perform a lookup in the index and the time to retrieve the answer from the cracker column; (ii) "Indexing Total Time" (ITT) is the total time to crack the column and insert new nodes into the index; (iii) "Update Total Time" (UTT) is the total time to shuffle data into the cracker column and update the index; (iv) "Pruning Time" (PT) is the time to prune the data. The total execution time of a query is the sum of the four measurements.

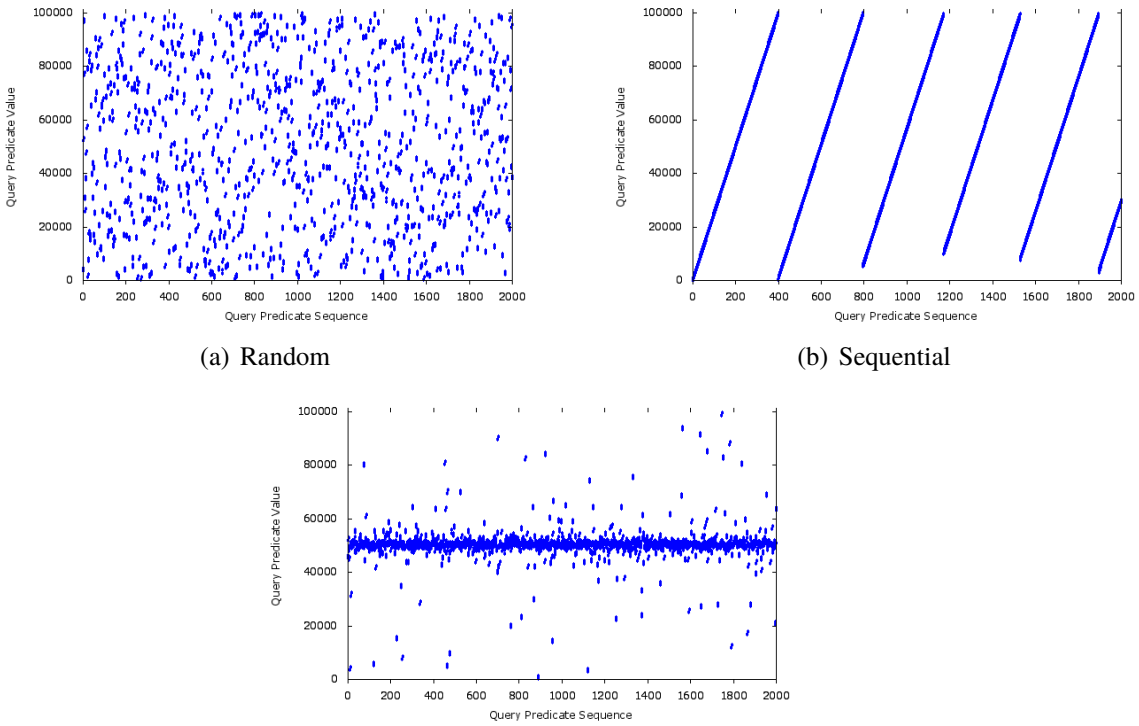
Our experiments focus on the Standard Database Cracking algorithm, because the results of the SPST do not favor any specific cracking approach considering read only and mixed workloads. This happens since hot/cold data depends on the workload and not on the running cracking algorithm. Initially, considering mixed workloads it seems that the Hybrid Cracking does not comply with the pruning strategy, since it is always cracking more pieces to achieve a faster convergence to a full index, but our currently pruning strategy guarantees that the tree is pruned only when the gains in updates surpass the losses in cracking. That way we are able to achieve similar results for all the proposed cracking strategies in the literature.

Setup. We implemented our data structure and performed all the experiments using the database cracking simulator[1] presented by [31]. 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.

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 5.1. 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

[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`

by Figures 5.1(a), 5.1(b), and 5.1(c). The skewed workload is generated by the zipf's law with α equals to 2.0. All presented workloads have selectivity equal to 1.



(a) Random

(b) Sequential

(c) Skewed

Figure 5.1: Workload Patterns

5.1 Select Operator

In this section we analyze the read operations with the three workload patterns. We focus our analysis on the index lookup time for both "Query Total Time" and "Indexing Total Time", while we focus on the index update time for the "Indexing Total Time" (i.e., Index lookup and maintenance times). In particular, we observed the IPC and cache misses (L1/2/3). We consider as the best cache-efficient data structure the one with the highest IPC and smallest number of cache misses.

To analyze IPC and Cache misses, we used the native tools provided by Apple for the MacOS Sierra operating system. For IPC is considered the predefined IPC Formula in the Instruments app for Mac OS.

For cache misses (L1/2/3), we captured the following respective events[2]:

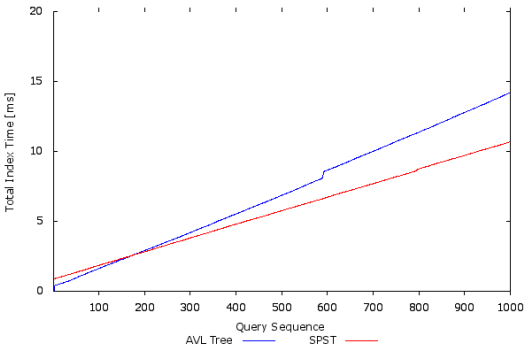
- The number of instruction misses in the L1 cache, option: *ICache.Misses*
- All requests that miss L2 cache, option: *L2_RQSTS.Miss*;

[2]The event counters are part of the "Intel 64 and IA-32 Architectures Software Developer's Manual" for our CPU model.

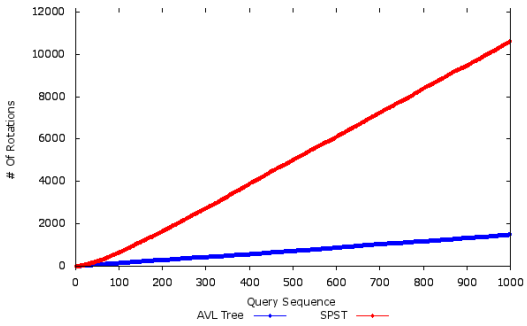
- Miss in last-level (L3) cache, option:
  Mem_Load_Uops_Retired.L3_Hit_ps.

The rest of this section presents an analysis of techniques to reduce tree rotations and an analysis of the total indexing time in a scenario with a skewed workload using a stream of 10^6 queries and a selectivity of 10^{-8}.

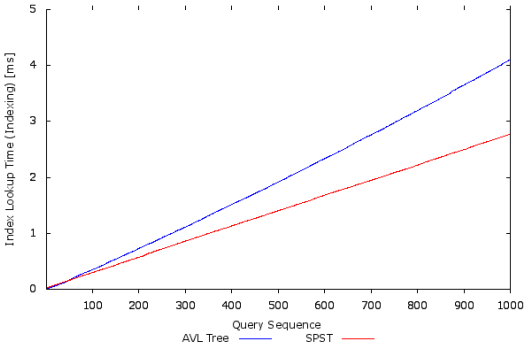
5.1.1 Random Pattern



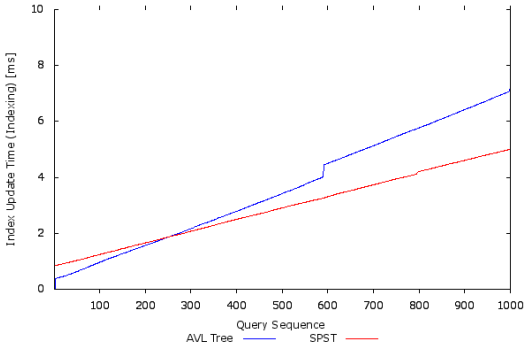
(a) Total Lookup and Maintenance Index Time



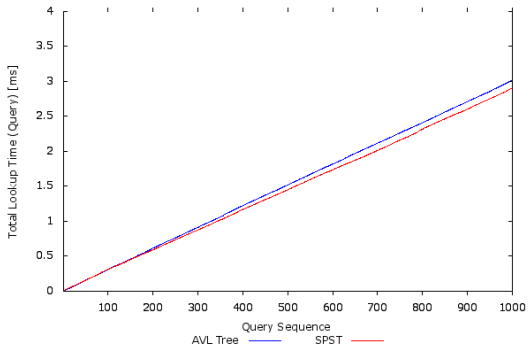
(b) Number of Trees Rotations



(c) Lookup Time While Indexing



(d) Insertion Time



(e) Lookup Time

Figure 5.2: Cracker Index in Random Workload

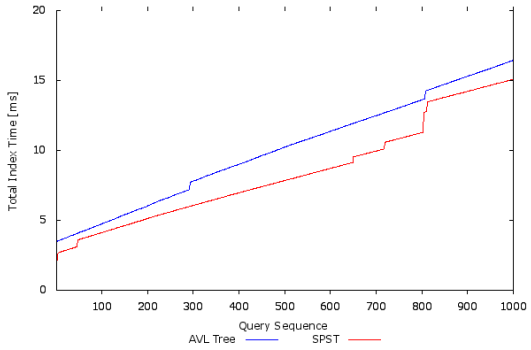
Tree	L1	L2	L3	IPC
AVL	1,108,508,606	4,972,838,130	252,404,784	1.094
SPST	267,097,844	3,957,313,535	135,615,510	1.385
Improvement Ratio	400%	25%	90%	22%

Table 5.1: Cache Misses and IPC in Random Workload

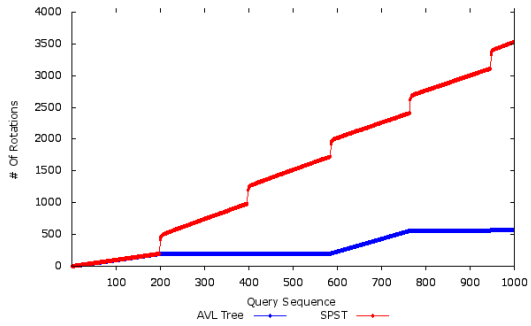
Figure 5.2(a) depicts the accumulated index lookup and maintenance time for the query stream in the random workload. The AVL Tree was faster than the SPST for the first 180 queries, because the random workload demanded a higher number of rotations in the SPST to settle down the range pattern close to the root. With more incoming queries, the splay tree started to leverage the cached nodes from the root running the $1,000^{th}$ query 21.5% faster than the AVL Tree. The number of rotations in the SPST was still higher than the AVL, as depicted by Figure 5.2(b). However, the rotations in the SPST presented lower 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 all over the index to rebalance the tree with many unnecessary tree nodes polluting the cache (see Table 5.1). Figure 5.2(c) depicts the cost reduction achieved by the SPST since it is already rotated for the most accessed nodes. Figure 5.2(d) presents that even with fewer rotations the AVL Tree quickly achieve a higher overall cost after the 250^{th} query, due to rotations occurring at leaves. This shows the SPST is able to leverage the cached nodes and match the lookup time of the AVL Tree (see Figure 5.2(e)).

5.1.2 Sequential Pattern

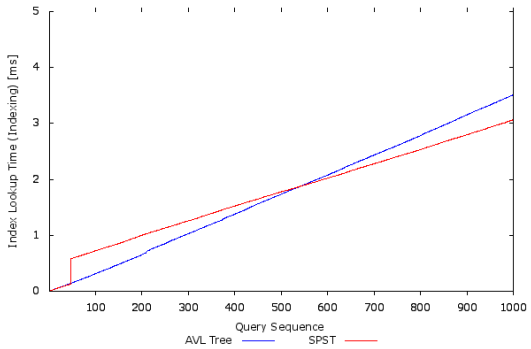
The sequential pattern was the worst case scenario for the SPST, but still the SPST reduced the total index time in 7% compated to the AVL Tree at the $1,000^{th}$ (see Figure 5.3(a)). The worst case scenario is a result of many changes in the range predicate of the sequential pattern that require splaying nodes from the leaves. Over time the SPST mitigates these rotations with 16.9% less L1 cache misses compared to the AVL (see Table 5.2). However, we see the result of these predicate changes in small jumps in the rotations, as depicted by Figure 5.3(b). The sequential pattern produces the least number of rotations compared to the other workloads, because of the splaying operations: when a node is splayed, its children also go towards the root. With incoming sequential scans, lookups find the nodes already rotated up in the tree and then require less rotations to reach the root. Figure 5.3(c) depicts that the SPST only achieves a faster response time around the 550^{th} query, meaning that it takes more time to find Hot Data than the Random pattern. Figure 5.3(d) presents the sudden changes in pattern around the 650^{th} query with heavy insertions. These pattern changes are the main reason for the worst result of the SPST. But, still the SPST was capable of leveraging the cached nodes to mitigate the cost of the rotations (see Figure 5.3(e)).



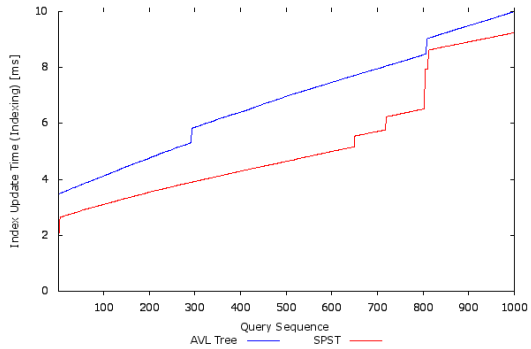
(a) Total Lookup and Maintenance Index Time



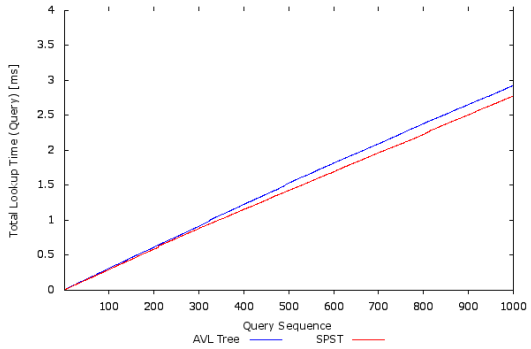
(b) Trees Rotations



(c) Lookup Time While Indexing



(d) Insertion Time



(e) Lookup Time While Querying

Figure 5.3: Cracker Index in Sequential Workload

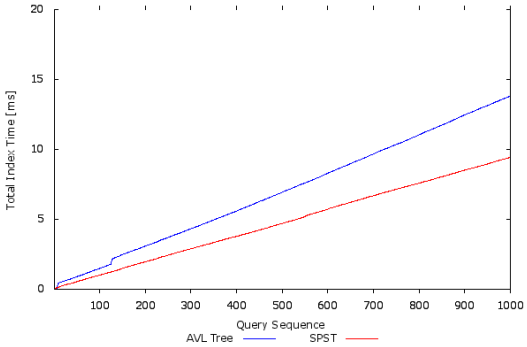
Tree	L1	L2	L3	IPC
AVL	855,925,856	10,890,330,930	412,469,096	1.234
SPST	711,228,747	10,479,242,239	399,344,564	1.263
Improvement Ratio	20%	4%	4%	3%

Table 5.2: Cache Misses and IPC in Sequential Workload

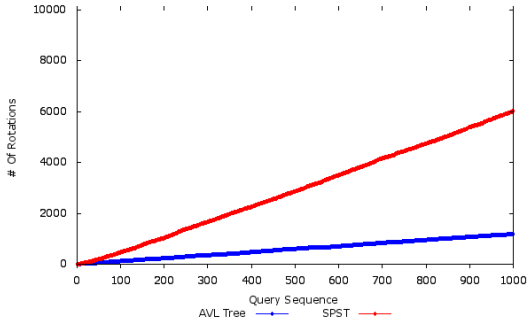
5.1.3 Skewed Pattern

The skewed pattern was the best case scenario for the SPST, because queries focus on "Hot Data". This favors the SPST, which reduced cache misses and boosted IPC improving response time overall. The SPST reduced the total index time in 37% and showed 55.2% less L1

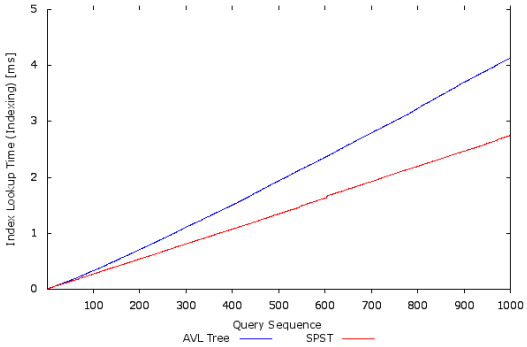
cache misses compared to the AVL Tree at the $1,000^{th}$ (see Figure 5.4(a) and Table 5.3). This also favored the update of the index with the SPST faster than the AVL as rotations are close the root, although the SPST rotates more nodes (see Figures 5.4(b) and 5.4(d)). The skewed workload demands around 100 queries to the SPST cache Hot Data and then settle down the range pattern close to the root (see Figure 5.4(c)). This shows the benefit of the splayings that make Hot Data mostly fitting in cache. Obviously, this benefit extends to the lookup for querying with the SPST outperforming the AVL (see Figure 5.4(e)).



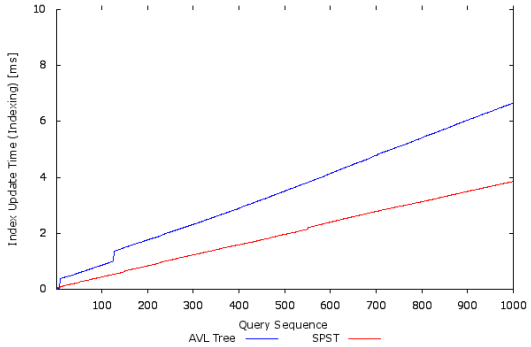
(a) Total Lookup and Maintenance Index Time



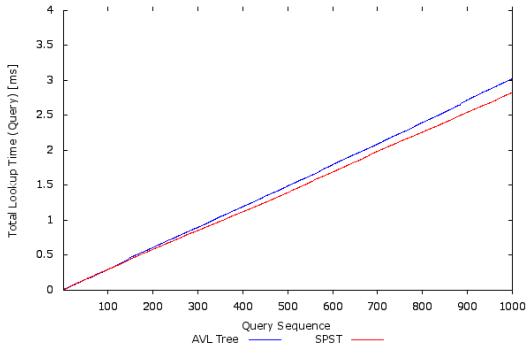
(b) Trees Rotations



(c) Lookup Time While Indexing



(d) Insertion Time



(e) Lookup Time While Querying

Figure 5.4: Cracker Index in Skewed Workload

Table 5.3: Cache Misses and IPC in Skewed Workload

Tree	L1	L2	L3	IPC
AVL	573,854,301	3,800,678,199	176,536,452	1.160
SPST	256,760,334	3,780,063,118	128,213,328	1.600
Improvement Ratio	250%	0.5%	40%	28%

5.1.4 High Selectivity

In sections 5.1.1, 5.1.2 and 5.1.3 the experiments were conducted with a stream of 1,000 queries and selectivity of 1. In those scenarios, the cracking time is usually much higher than the lookup time, making the gains in lookup time meaningless when considering the total time. However, as the cracker index converges to a full index, the lookup time becomes higher than the cracking time. To validate if the gains persist when the lookups become more expensive than the cracking time, we performed a high selectivity experiment with a stream of 10^6 queries. By high selectivity, we mean the predicates of a range query are located close together and the index refinement per query is small. We choose the Skewed scenario to validate if it still persists as the best case scenario of the SPST when changing the selectivity (see Figure 5.5).

At the $10^6 th$ query, response time with the SPST was 31% better than the AVL. The decrease of response time when compared to Figure 5.4(a) is due to the increasing number of unused nodes being stored at the AVL tree. This experiment depicts that as the partial AVL index converge to the full index the lookup time starts to get bigger and the differences achieved in index lookup and maintenance time start to have a higher impact in the overall query execution time.

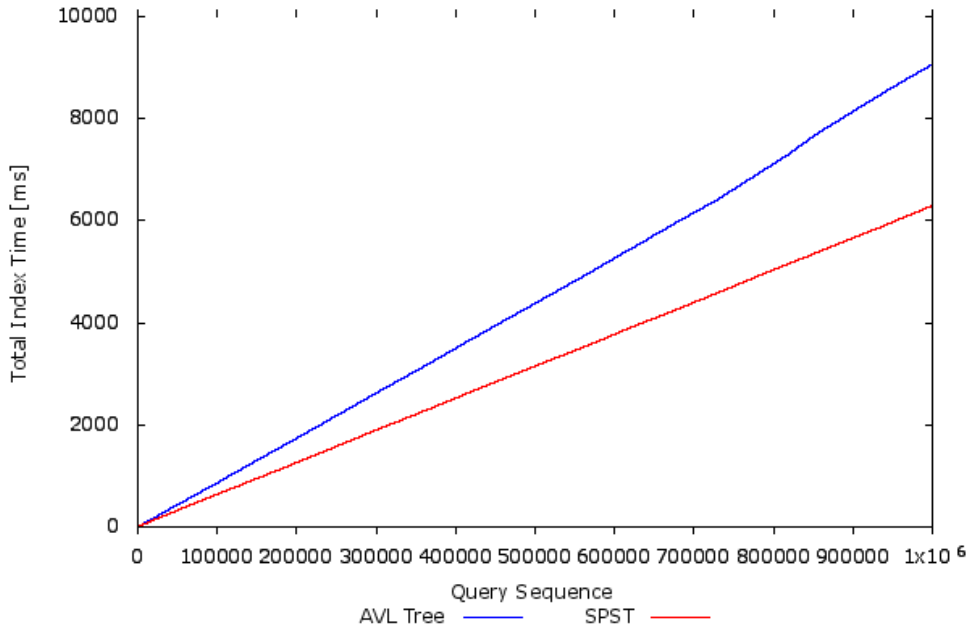
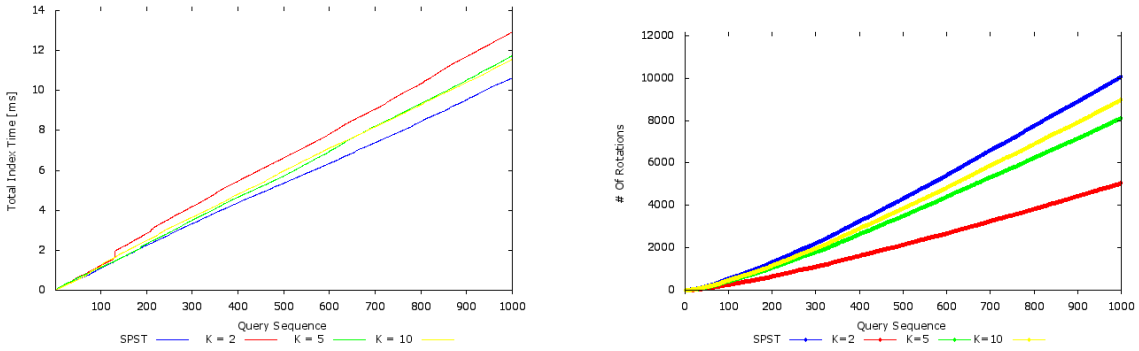


Figure 5.5: Indexing Cost in High Selectivity Scenario with a stream of 10^6 queries.

5.2 Reducing Rotations with K-Splaying

In sections 5.1.1, 5.1.2 and 5.1.3 all experiments presented the SPST with a higher number of rotations when compared to the AVL Tree. In this experiment we run the SPST with the K-Splaying technique to see if reducing rotations can bring a better response time. Figure 5.6 depicts the results for K-Splaying with 3 different configurations: $k = 2$, $k = 5$ and $k = 10$. We tested the K-Splaying in the "Random Workload" scenario, because it presented the largest number of rotations for the SPST.

As expected, the K-Splaying was able to reduce the number of rotations (see Figure 5.6(b)), however, when we apply K-Splaying those rotations start to happen more far away from the root causing higher costs (see Figure 5.6(a)). We conclude that the K-Splaying technique should only be applied in situations where updating pointers impact in response time (e.g., in hardware with slow memory access). This impact is low in the case of the SPST, because most of the rotations happen in cache.



(a) K-Splaying with different K setups.

(b) Number of rotations by K.

Figure 5.6: Reducing Rotations with K-Splaying

5.3 Update Operator

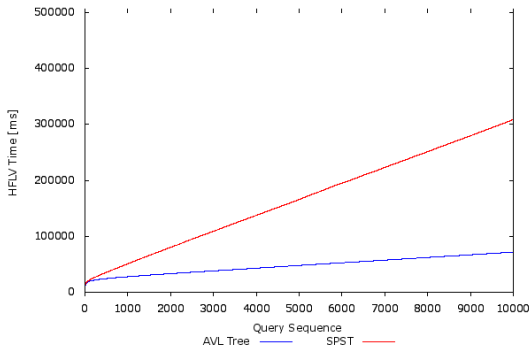
For write operations, we considered two update scenarios: low frequency high volume updates (i.e., LFHV), and high frequency low volume updates (i.e., HFLV). In the LFHV scenario we execute 1,000 queries followed by a batch of 1,000 updates. In the HFLV scenario we execute 10 queries followed by a batch of 10 updates. We execute the random pattern for queries and updates. 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 focus our analysis on the sum of total cracking time (ITT), index update time (UTT) and index pruning (PT), because these metrics are related to updates affected by pruning the SPST. Our analysis also extends to the total tree size and the number of rotations done by the AVL, the traditional Splay Tree (no pruning) and the SPST. All the update experiments run a stream of 10,000 queries.

We start our analysis by motivating the use of the standard deviation pruning, comparing the situations with and without the deviation and analyzing the cost breakdown of all the elements. Later on, we perform the experiments with the sequential and skewed workloads and analyze the total indexing time of them. We defined empirically a standard deviation threshold $p = 0.2$ milliseconds for HFLV and $p = 200$ milliseconds for LFHV. These values differ because for HFLV we only analyzed the standard deviation for 10 queries prior to a batch update, while for LFHV 1,000 queries are analyzed.

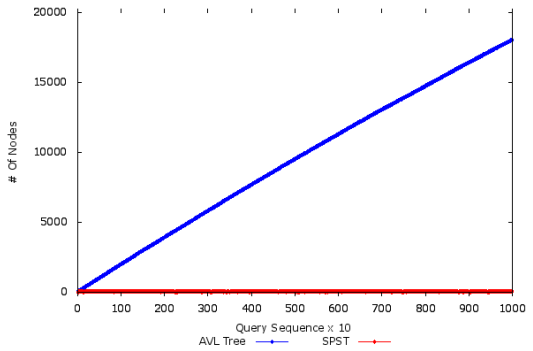
In our implementation and experiments, we stick to the MCI algorithm, because it is more sensitive to pruning as the constant shuffling requires more adjustments to the cracker index.

5.3.1 Random Pattern Without Standard Deviation

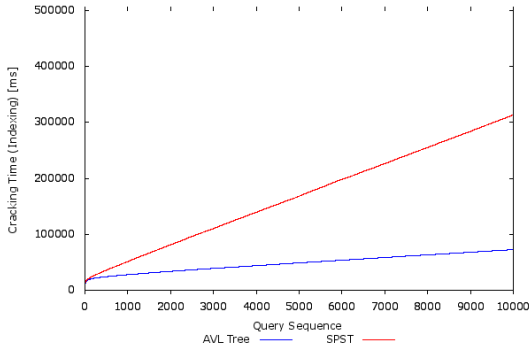
In this section, we analyze the index update activity based only on the frequency of the incoming queries.



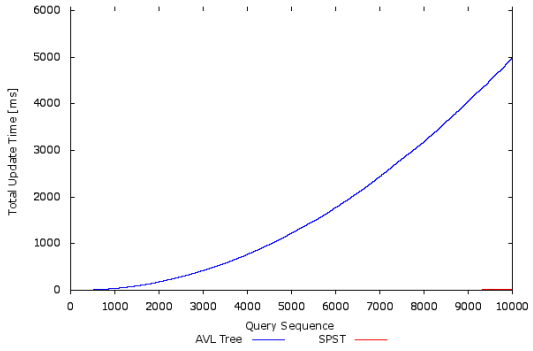
(a) HFLV



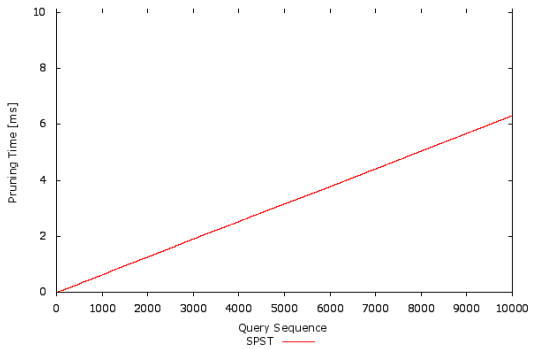
(b) Size



(c) ITT



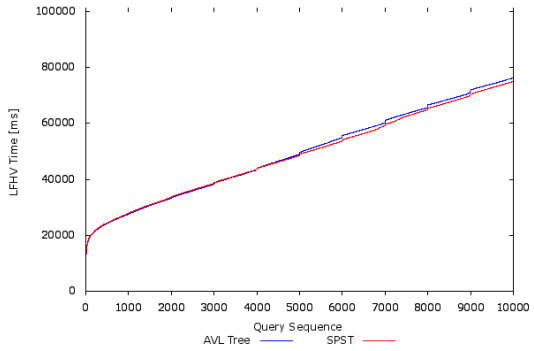
(d) UTT



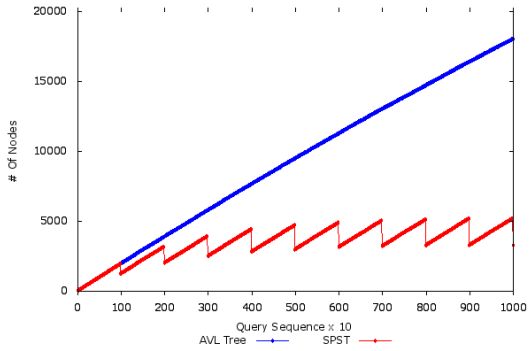
(e) PT

Figure 5.7: Cost Breakdown of Random Workload Without Standard Deviation in HFLV

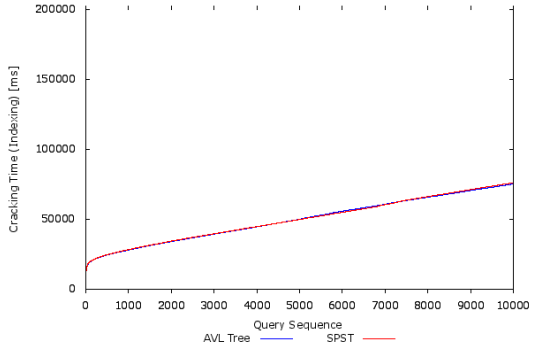
Figure 5.7(a) depicts the cost breakdown of the total Indexing Time for the HFLV. Although the SPST presented an update time orders of magnitude faster than the AVL (See Figure 5.7(d)), the SPST presented a total response time 4x worse than the AVL Tree. This happens because of the pruning frequency at every 10 queries resulting in a tree with a small number of nodes and a frequent re-cracking activity (See Figures 5.7(b) and 5.7(c)).



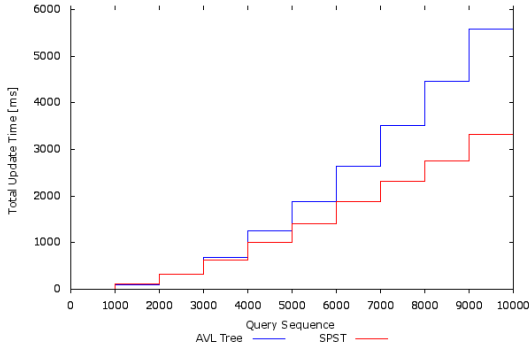
(a) LFHV



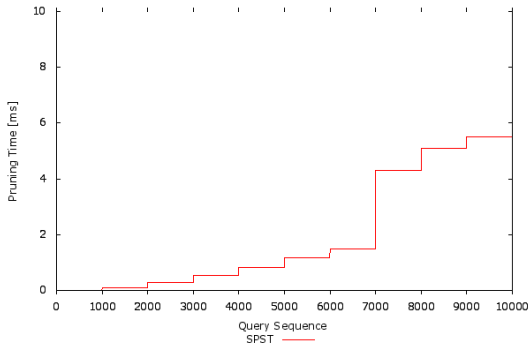
(b) Size



(c) ITT



(d) UTT

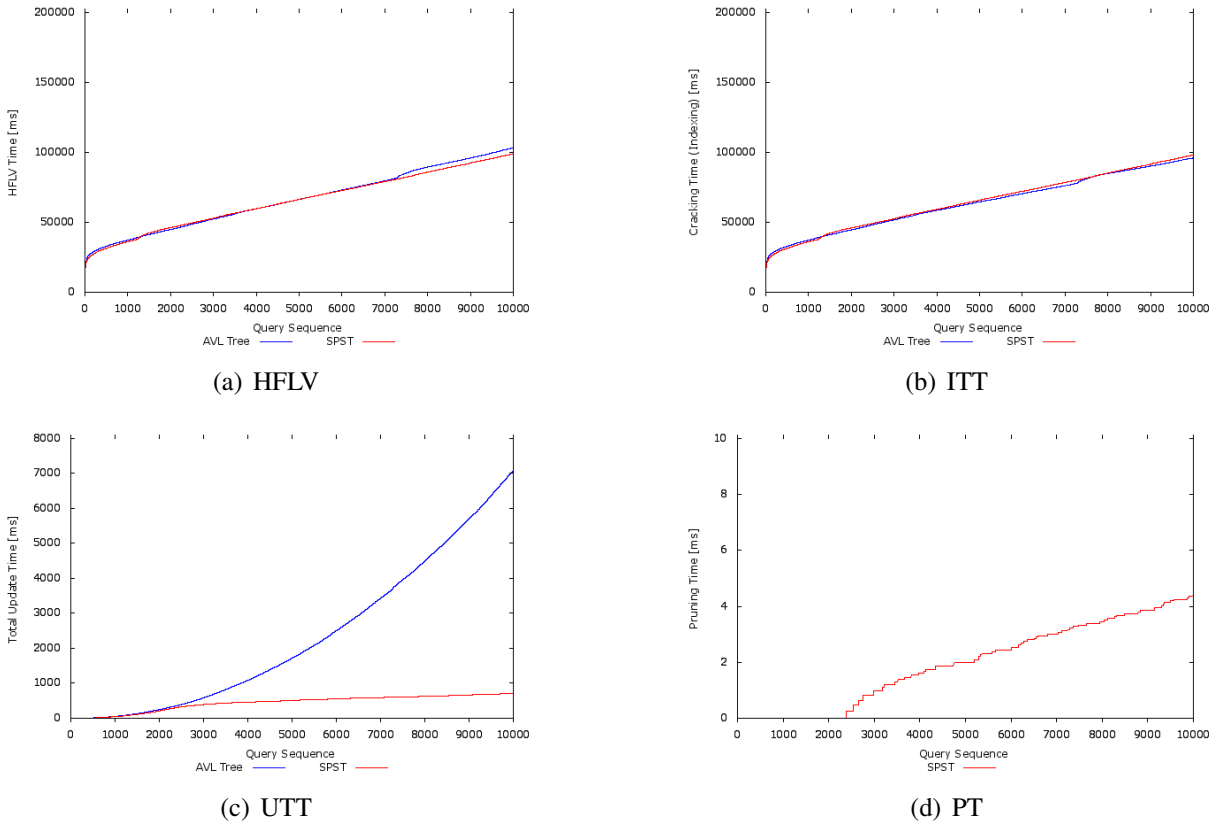


(e) PT

Figure 5.8: Cost Breakdown of Random Workload Without Standard Deviation in LFHV

Figures 5.8(a) and 5.8(c) depict the cost breakdown of the total Indexing Time and lookup for the LFHV. The SPST presented a better response time in 4% when compared to the AVL Tree, because with less frequent re-cracking, the biggest cost comes from the index update 50% cheaper than the AVL Tree. In this experiment the SPST pruned itself at every 1,000 queries resulting in a tree with 75% less nodes (See Figures 5.8(b) and 5.8(e)). Moreover, the index update costs shrank with a smaller tree (See Figure 5.8(d)).

5.3.2 Random Pattern



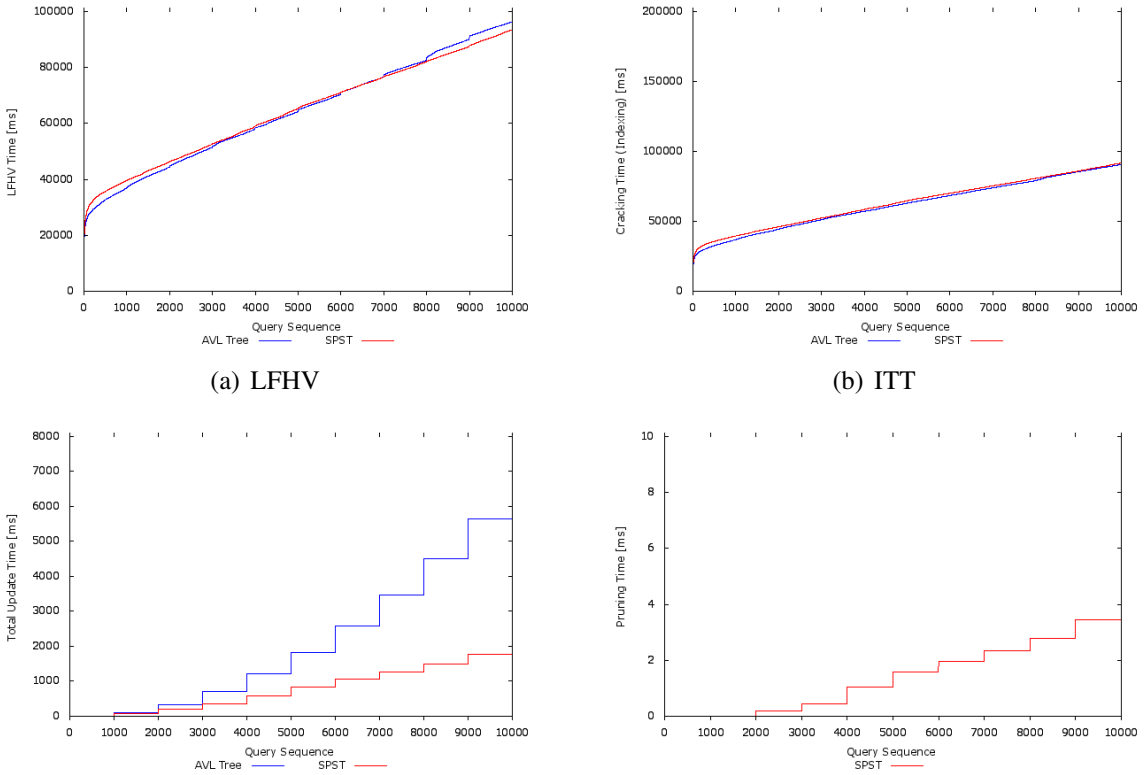
(a) HFLV (b) ITT

(c) UTT (d) PT

Figure 5.9: Cost Breakdown of Random Workload With Standard Deviation in HFLV

Now we discuss the cost breakdown of the total Indexing Time for the HFLV (See Figure 5.9(a)). The SPST presented a total response time 5% faster than the AVL Tree and matched the response time of the AVL for HFLV. This happens because the SPST took the standard deviation of cracking time in consideration starting to prune only around the $2,300^{th}$ query and then pruning again only if the cracking does not stabilize. But, still the pruning happened more constantly than with LFHV. The result was 10x faster updates, 3% slower cracking and pruning with an additional cost of 4ms, respectively depicted by Figures 5.9(b), 5.9(c) and 5.9(d).

The results of the LFHV are similar to the HFLV, but with the SPST starting to prune at the $2,000^{th}$ query and with a frequency of 1,000 queries if the cracking does not stabilize. These results are interesting, because they show the standard deviation benefits both update frequencies in the random pattern. The SPST was always cheaper to update, as depicted by Figure 5.10, with 4x faster updates, although cracking was 1% slower and pruning had an additional cost of 3.8ms.

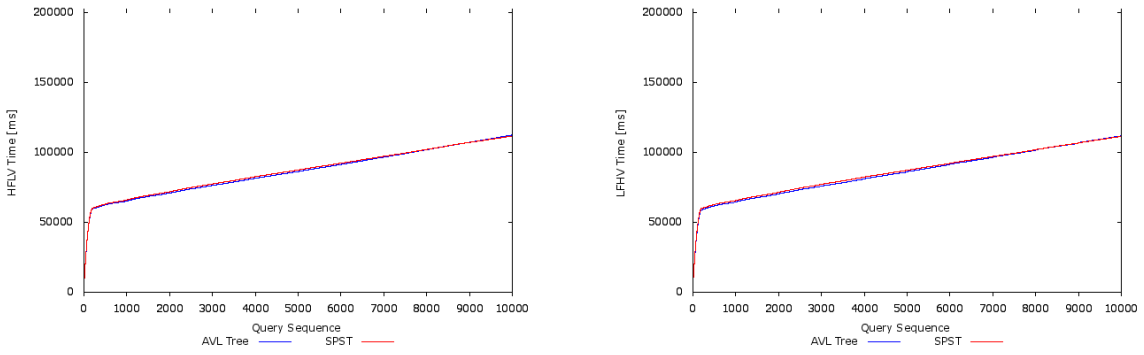


(a) LFHV (b) ITT

(c) UTT (d) PT

Figure 5.10: Cost Breakdown of Random Workload With Standard Deviation in LFHV

5.3.3 Sequential Pattern



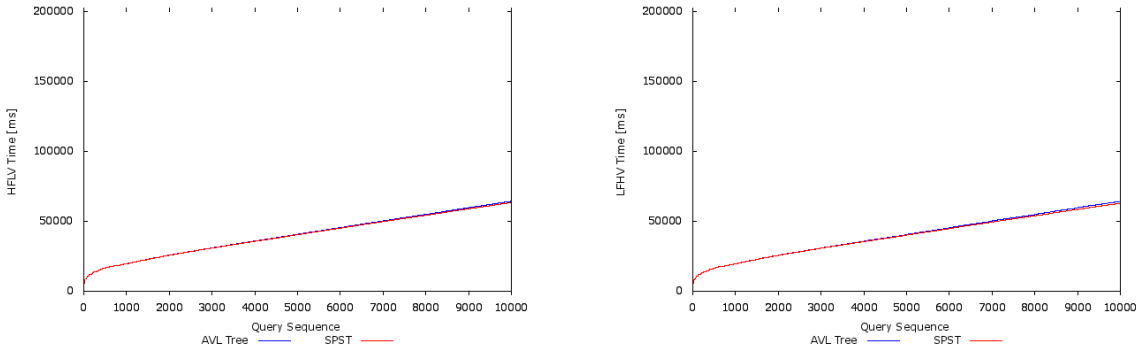
(a) HFLV (b) LFHV

Figure 5.11: Sequential Workload with Updates

Figure 5.11 depicts the Sequential Workloads with updates. Figure 5.11(a)) depicts the results in HFLV scenario, the SPST presents a total response time 1% faster than the AVL Tree. Figure 5.11(b) depicts the results in LFHV scenario. The SPST presents a total response time 1% faster than the AVL Tree, for both the HFLV and the LFHV, the SPST finishes the experiment with 25% of the AVL size. This being the result with least difference between the SPST and the AVL. This happens since the sequential workload visits all the data sequentially, that way,

data that is pruned usually ends up being cracked again. For sequential workloads although the update costs decrease in 3x the cracking that needs to be done again hides this benefit.

5.3.4 Skewed Pattern



(a) HFLV (b) LFHV

Figure 5.12: Skewed Workload with Updates

Figure 5.12 depicts the Skewed Workloads with updates. Figure 5.12(a)) depicts the results in HFLV scenario, the SPST presents a total response time 2% faster than the AVL Tree. Figure 5.12(b) depicts the results in LFHV scenario. The SPST presents a total response time 3% faster than the AVL Tree, for both the HFLV and the LFHV, the SPST finishes the experiment with 13% of the AVL size. In a skewed workload, pruning does not have high impact in the total cost. This happens because we only keep the highly skewed data in the index, that way, when we prune the cold data, cracking become much more expensive when we have to crack pieces that are far away from our hot data.

Chapter 6

Conclusion and Future Work

Database Cracking, become in the last decade a mature field of research. It is an effective technique for adaptive indexing, by creating its indexes partially and adaptively as the result of query processing. The interest in database cracking and adaptive index has pushed researchers to spend efforts to optimize them, with other cracking algorithms. Our contribution is intended to be another brick in this optimization effort. Our contribution is based on the hypothesis that we can optimize the cracker index by caching "Hot Data" at the root of the tree, clustering it to boost access and pruning unused data at the leaves to boost updates. In this work we presented our new data structure, the SPST, and demonstrated that the SPST is able to cache the most used data, achieving higher IPC and lower Cache Misses and to prune cold data, achieving a small storage footprint and boosting updates. Also, through experimentation we noticed that the SPST number of rotations is higher than the AVL Tree, however they present a minor impact in response time, since those rotations happen close to the root of the tree. Also, we noticed that using the SPST in the beginning does not achieve high results in the overall query, but as the index converges to a full index, changing the index structure to a cache conscious one starts to achieve a higher difference and that pruning also achieve higher results when we have bigger batches of updates. Future work is required in the following:

- Implement the SPST into MonetDB cracking implementation [14] and repeat the same experiments performed in the simulator.
- Other strategies to reduce rotations may optimize the structure even further (e.g., freezing the tree depending on the standard deviation of index lookups).
- Other pruning strategies (e.g, use a size threshold when having limited memory).
- Explore how the SPST fits with cracker joins [22] since they suffer from high administration costs.
- Compare the SPST with other index structure regarding miss and hit of the Translation Lookaside Buffer.
- 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.
- Explore how the SPST fits with holistic indexing [30] (e.g., pruning nodes in order to make the distance between two consecutive nodes fit in L1.)

Bibliography

[1] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, Samuel Madden, et al. *The design and implementation of modern column-oriented database systems*. Now, 2013.

[2] Timo Aho, Tapio Elomaa, and Jussi Kujala. *Reducing Splaying by Taking Advantage of Working Sets*, pages 1–13. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[3] Anastassia Ailamaki, David J DeWitt, and Mark D Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 11(3):198–215, 2002.

[4] Susanne Albers and Marek Karpinski. Randomized splay trees: theoretical and experimental results. *Information Processing Letters*, 81(4):213–221, 2002.

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

[6] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. Concurrent cache-oblivious b-trees. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 228–237, 2005.

[7] Jon Louis Bentley. Multidimensional binary search trees in database applications. *IEEE Transactions on Software Engineering*, 4:333–340, 1979.

[8] Nicolas Bruno. *Automated Physical Database Design and Tunning*. CRC-Press, 2011.

[9] Surajit Chaudhuri and Vivek Narasayya. Autoadmin "what-if" index analysis utility. *ACM SIGMOD Record*, 27(2):367–378, 1998.

[10] Bernard Chazelle and Leonidas J Guibas. Fractional cascading: A data structuring technique with geometric applications. In *International Colloquium on Automata, Languages, and Programming*, pages 90–100. Springer, 1985.

[11] Douglas Comer. The difficulty of optimum index selection. *ACM Transactions on Database Systems (TODS)*, 3(4):440–445, 1978.

[12] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[13] George P Copeland and Setrag N Khoshafian. A decomposition storage model. In *ACM SIGMOD Record*, volume 14, pages 268–279. ACM, 1985.

[14] CWI. Monetdb. `www.monetdb.org`, 2016.

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

[16] Martin Fürer. Randomized splay trees. *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pages 903–904, 1999.

[17] Goetz Graefe, Felix Halim, Stratos Idreos, Harumi Kuno, and Stefan Manegold. Concurrency control for adaptive indexing. *Proceedings of the VLDB Endowment*, 5(7):656–667, 2012.

[18] Goetz Graefe, Stratos Idreos, Harumi Kuno, and Stefan Manegold. Benchmarking adaptive indexing. *Technology Conference on Performance Evaluation and Benchmarking*, pages 169–184, 2010.

[19] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland HC Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *VLDB*, 5(6):502–513, 2012.

[20] Ralf Hinze et al. Constructing red-black trees. In *Proceedings of the Workshop on Algorithmic Aspects of Advanced Programming Languages, WAAAPL*, volume 99, pages 89–99, 1999.

[21] Pedro Holanda and Eduardo Cunha de Almeida. Spst-index: A self-pruning splay tree index for caching database cracking. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT.*, 2017.

[22] Stratos Idreos. *Database Cracking: Towards Auto-tuning Database Kernels*. PhD thesis, Harvard, Cambridge - United States, 2013.

[23] Stratos Idreos, Martin L Kersten, and Stefan Manegold. Updating a cracked database. In *SIGMOD*, pages 413–424. ACM, 2007.

[24] Stratos Idreos, Martin L Kersten, and Stefan Manegold. Self-organizing tuple reconstruction in column-stores. *SIGMOD*, pages 297–308, 2009.

[25] Stratos Idreos, Martin L Kersten, Stefan Manegold, et al. Database cracking. In *CIDR*, volume 3, pages 1–8, 2007.

[26] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores. *VLDB*, 4(9):586–597, 2011.

[27] John Jenkins, Isha Arkatkar, Sriram Lakshminarasimhan, David A. Boyuka II, Eric R. Schendel, Neil Shah, Stéphane Ethier, Choong-Seock Chang, Jackie Chen, Hemanth Kolla, Scott Klasky, Robert B. Ross, and Nagiza F. Samatova. ALACRITY: analytics-driven lossless data compression for rapid in-situ indexing, storing, and querying. *Trans. Large-Scale Data- and Knowledge-Centered Systems*, 10:95–114, 2013.

[28] Eric K Lee and Charles U Martel. When to use splay trees. *Software-Practice and Experience*, 37(15):1559–1576, 2007.

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

[30] Eleni Petraki, Stratos Idreos, and Stefan Manegold. Holistic indexing in main-memory column-stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1153–1166. ACM, 2015.

[31] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. The uncracked pieces in database cracking. *VLDB*, 7(2):97–108, 2013.

[32] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

[33] Stefan Sprenger, Steffen Zeuch, and Ulf Leser. Cache-sensitive skip list: Efficient range queries on modern cpus. *VLDB*, 2016.

[34] Andrew Chi-Chih Yao. On random 2–3 trees. *Acta Informatica*, 9(2):159–170, 1978.

[35] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. Cooperative scans: Dynamic bandwidth sharing in a dbms. In *VLDB*, pages 723–734, 2007.