Incremental Branching Adaptive Radix Tree

Heba El-Fadly
Computers and Automatic Control Dept.
Faculty of Engineering
Tanta University
Tanta, Egypt
heba@innotech.com.eg

ElSayed Sallam
Computers and Automatic Control Dept.
Faculty of Engineering
Tanta University
Tanta, Egypt
sallam@f-eng.tanta.edu.eg

Mohamed Shoaib
Computers and Automatic Control Dept.
Faculty of Engineering
Tanta University
Tanta, Egypt
mohamed.shouaib@f-eng.tanta.edu.eg

***Abstract*— Adaptive indexing has been an area of active research in recent years. Its main concept is to create and maintain indexes adaptively and incrementally based on the incoming workload as a part of the query execution process. Database cracking is the first introduced applicable adaptive indexing paradigm. Despite the effectiveness and lightweight of database cracking, it offers slower lookups when it is compared to modern main memory index structures like adaptive radix tree (ART). ART is a recently proposed main-memory index structure which is designed to be space-efficient yet offers high performance by adapting its internal node size based on the count of keys stored in it.**

In this paper, we presented IBART (incremental branching adaptive radix tree), a hybrid indexing technique that takes the main concept of database cracking and applies it to ART, generating an adaptive index in terms of its creation and maintenance and also its internal nodes size. For systems of dynamic nature and unpredictable workload, IBART is proven to be more convenient than ART.

Keywords—adaptive indexing; database cracking; adaptive radix tree; ART; IBART

I. INTRODUCTION

Physical design affects the database management systems performance significantly. Over the years, many approaches have been proposed that aim to boost the efficiency of the transactions execution process. Indexing is one of these techniques, which effectively enhance the database querying response time. The process of deciding which index to be created or dropped requires deep knowledge of database architectural design and the system workload. For the systems of predictable workload and relatively small number of promising indexes, the database administrator (DBA) is responsible for monitoring the system performance and adapting its physical design accordingly.

As the database logical model becomes more complex, the number of candidate indexes increases and it gets very harder to the DBA to decide which ones to be created or dropped. Accordingly, there was a crucial need for some kind of automated procedures to make it easier for the DBA to make his decision. Database monitoring is one of these procedures that simplifies the task of the DBA significantly [1], [2]. Its main core is monitoring incoming queries and response time, and then it produces statistics and recommends a set of physical design updates. Next, the DBA role is simply to make his decision based on the knowledge provided to him.

As the workloads became more dynamic and unpredictable, tuning the indexes in response to monitoring the incoming requests suffers from multiple weaknesses [3], [4]. First, after creating an index based on the monitoring tools recommendation, the workload may change such that the created index is not useful anymore. Second, the executed queries during the monitoring phase neither get benefits from nor aids index creation efforts. Last, but not least, the traditional indexes contains all values of the concerned column, while some of them are frequently accessed and some never.

Moreover, the data is growing progressively in volume, velocity, and diversity. For large volume database, changing the physical design by any way is considered as a "heavyweight" operation [5]; for example, it takes too much time and requires many resources to create or update an index of a big data set.

Adaptive indexing was introduced to overcome the drawbacks of the traditional indexing techniques. It aims to maintain the changes in the physical design of the index in a continuous, incremental, adaptive, and partial way on demand [6]. Database cracking is the first introduced applicable algorithm of adaptive indexing[7]. The main concept of database cracking is creating and maintaining indexes adaptively and incrementally as a part of the query execution process, considering each incoming request as a suggestion that will be used to refine the created index. The cracking process is very similar to quick sort, where each query performs a partitioning step and the pivot is chosen based on the query predicates. By this way, the indexing creation process is distributed over several subsequent queries and each query benefits from the preceding ones.

Despite the effectiveness and lightweight of database cracking, it offers slower lookups when it is compared to modern main memory index structures [8]. Recently, V. Leis et al. proposed Adaptive radix tree (ART), which is a main-memory index structure designed to be space-efficient yet offers high performance[9]. Based on the experimental evaluation proposed by F. M. Schuhknecht et al., ART was proved to have 1.8 times faster response compared to standard cracking after 1000 queries and 3.6 times faster after 1M queries[8].

In this paper, we introduce incremental branching adaptive radix tree (IBART), a hybrid indexing technique that takes the benefits of ART and constructs it incrementally and adaptively based on the database cracking concept.

The rest of paper sections are organized as follows. The next section discusses related work. Section III presents the incremental branching adaptive radix tree. Section IV describes experimental results and evaluations. Finally, Section V concludes and the future work.

II. Related Work

Scientific researches on database physical design started to appear as early as 1974[5]. Materialized views[10], vertical partitioning[11], horizontal partitioning[12], and indexing[13] are the preliminary disciplines of database physical design that was proved to have a significant impact on the DBMS performance.

Several indexing techniques methods have been introduced in the literature. Gani et al. presented a state-of-the-art comparing and analyzing 48 indexing technologies [14].

One of the vital research topics about indexing is the procedure of deciding which indexes to create and when to create them[15]. Adaptive indexing was firstly introduced by Idreos et al. at 2007[7]. They aim to dispense the role of DBA in the index creation process by generating a system that can adapt and tune itself, even in dynamically changing environments.

Database cracking is the first applicable adaptive indexing paradigm [7]. It pioneered the idea of incrementally creating indexes throughout query processing. It considers each incoming query as a hint that helps in maintaining and refining the created index and enhances the response time of the subsequent queries. Database cracking assumes no prior knowledge of the system workload. When column A is queried for the first time, a copy of the column is created called cracker column A_{CRK}. For each subsequent query, A_{CRK} is continuously partitioned into smaller and smaller pieces based on the incoming workload.

The cracking procedure is shown in figure 1. It is very similar to the quick-sort algorithm, where each query performs one partitioning step and the pivot is dynamically chosen based on the query predicates. For example, assuming the following query (A < V1), the cracked column A_{CRK} will be split into two partitions, one holding the values smaller than V1 and the other holding the values greater than V1.

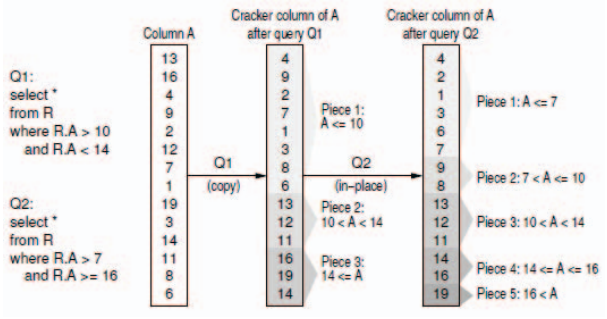


Fig. 1. Database Cracking [7]

A cracker index is created for each cracker column. The cracker index is an AVL-tree holding the information about the starting index of the cracker column pieces. This information is used to speed up the execution of the subsequent queries. For each query, first, the cracker index is searched to get the cracker pieces that contain the required range. Next, these pieces are cracked to get the exact range. Finally, the cracker index is updated to hold the information of the newly created cracks.

Since the premier cracking algorithm was introduced at 2007, several researches have proposed different adaptive indexing techniques. The same research group of the database cracking introduced sideways cracking, which enables processing queries on multiple attributes efficiently[16]. Graefe et al. proposed adaptive merging, which substitutes quick sort and AVL-tree of database cracking by merge sort and partitioned B-tree, respectively[17]. Next, a hybrid approach of database cracking and adaptive merging was introduced [18]. Stochastic cracking proposed a modified version of database cracking which avoids performance bottlenecks that may be caused due to cracking a column excessively[19].

For multicore systems, Alvarez et al. introduced three different adaptive indexing algorithms [20]. HAIL (Hadoop Aggressive Indexing Library) creates indexes adaptively as a side product of map reduce jobs [21]. Holistic Indexing was introduced by Petraki et al., which is a parallel version of database cracking [22]. Recently at 2017, SPST-Index was introduced [23]. It is an adaptive indexing technique that uses a splay tree instead of AVL-tree as a cracker index. Besides, Slalom was presented, which is an adaptive partitioning and indexing approach used to analyze raw data [24].

In order to assess the cracking algorithms, at 2015 Schuhknecht et al. introduced an experimental evaluation of multiple database cracking approaches and

compared them to multiple sorting algorithms and full index structures [8]. From the results of the experiment, the authors concluded that database cracking is a mature, lightweight, and repeatable algorithm. On the other hand, the results show that database cracking offers slower lookups when it is compared to modern main memory index structures. For example, ART was proved to have 1.8 times faster response compared to standard cracking after 1000 queries and 3.6 times faster after 1M queries.

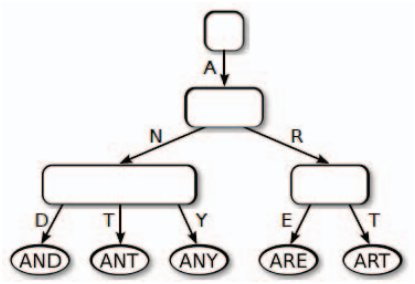


Fig. 2. Adaptive Radix Tree (ART) [9]

Adaptive radix tree (ART) is a modern main-memory indexing structure that was proposed at 2013[9]. It is a radix tree which takes the advantages of radix trees over the comparison based trees while, at the same time, it aims to overcome the excessive space-consumption caused by the spread of null pointers in the tree internal nodes, which most of radix trees suffer from. The key idea of ART is to adaptively use different node sizes based on the non-null child pointers as illustrated in Fig. 2.

ART has four types of nodes, Node4, Node16, Node48, and Node256, named based on the maximum number of children that they can hold.

Despite the proven efficiency of the ARTful index compared to other main-memory indexing techniques[9], like CSB [25], GPT [26], FAST [27], and even hash tables, it still creates indexes statically assuming a prior known, non-changing workload and idle time, which is not the case for most of nowadays systems. So, in this paper, we introduce incremental branching adaptive radix tree (IBART), which is an ART index which is constructed incrementally, partially and on demand based on the incoming workload, i.e. it inherits all the benefits of ART but constructs it based on the database cracking concept.

III. Incremental Branching Adaptive Radix Tree

This section discusses the Incremental Branching Adaptive Radix Tree (IBART) in details. First, a comparison between adaptive indexing and static indexing is presented. Next, the structure and process of IBART are introduced. Then, the proposed model algorithms are discussed. Finally, we make a quick comparison between ART and IBART and whether the second can be used as a replacement of the first.

A. Adaptive vs. Static Indexing

Some of the advantages of adaptive indexing over the static indexing are:

- Adaptive indexing requires no pre knowledge of the workload; it reacts dynamically to the workload changes.
- The effort required for creating an index is divided among all the queries that will benefit from this it; there is no need to penalize the first requester by creating the full index.
- As the index is created incrementally and on demand, the values that are not requested by any queries are not indexed.

B. New node structure: NodeCollapsed

The proposed model takes the advantages of the adaptive indexing and applies them to the adaptive radix tree. Fig. 3 illustrates the difference between the adaptive radix tree and the proposed incremental branching adaptive radix tree. The main idea is that the index tree branches adaptively and incrementally based on the incoming requests.

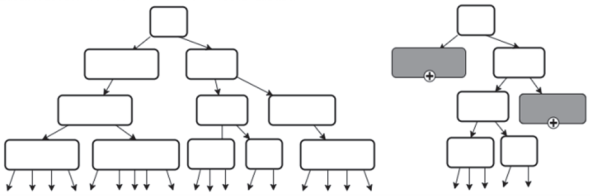


Fig. 3 ART (left) and our IBART (right)

To be able to build the tree incrementally, a new node type is proposed, named **NodeCollapsed**. The structure of the NodeCollapsed is a vector that holds all the values sharing the same partial key.

C. Querying IBART

The first time the column of concern is queried, the index tree of this column is constructed. The initial state of the radix tree is shown in Fig. 4. It consists of one root node, holding all the distinct most significant digits of the keys to be indexed, and each child of it points to a node of type NodeCollapsed, holding all the keys sharing the same most significant digit (MSD). The root node type is one of the well-known four ART node types (Node4, Node16, Node48, and Node256) and is chosen based on the count of its children. At this state, the tree is similar to the result of the first step of MSD based radix sort.

After the index tree initialization, the ART search algorithm is applied. The tree is traversed using the successive digits of the query key until a NodeCollapsed is encountered. At this point, the expansion process starts, which consists of three main steps. First, the keys contained in the NodeCollapsed are partitioned into groups based on the digit which the current tree level is

representing. Then the NodeCollapsed is replaced by an expanded ART node of suitable size that fits the generated key groups count. Finally, for each group, a child is added to the newly created ART node holding the group key digit and pointing to a node of type NodeCollapsed containing all the group keys.

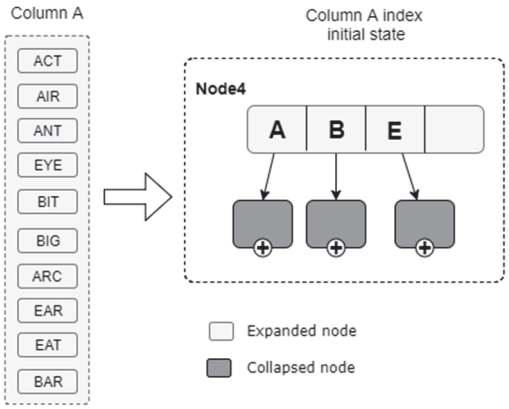


Fig. 4. IBART initial state

The previous three steps are repeated recursively until a leaf node or null pointer is reached. Fig. 5 represents the partially expanded index tree after searching for "BAR" key. This process is so-called incremental branching, and it can be simply thought as incremental MSD radix sort.

If another incoming request is searching for a previously queried key, it will found the key tree branch already expanded, so in this case the IBART index response similarly as the ART index. In contrast, if there is a collapsed node that was never reached by any query, it will not be expanded.

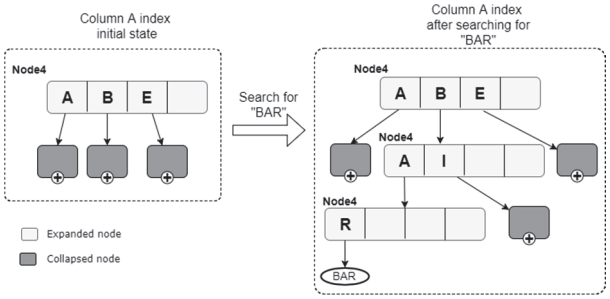


Fig. 5. IBART Querying Example

From the previous discussion, we conclude that the IBART index achieves the main two principles of adaptive indexing and database cracking[6]. First, each query works as a hint for the subsequent queries, and second, the non-queried keys are kept untouched.

D. Algorithms

We now present and discuss the pseudo codes of search and insert algorithms.

Search: Search algorithm pseudo code is shown in Fig. 6. It is the same search algorithm of the ART index [9] with no changes. The tree is traversed by the successive query key digits until a NULL pointer or a leaf node is found.

```
  search (node, key, depth)
 1 if node==NULL
 2   return NULL
 3 if isLeaf(node)
 4   if leafMatches(node, key, depth)
 5     return node
 6   return NULL
 7 if checkPrefix(node,key,depth)!=node.prefixLen
 8   return NULL
 9 depth=depth+node.prefixLen
10 next=findChild(node, key, depth)
11 return search(next, key, depth+1)
```

Fig. 6. Search Algorithm[9]

To find the next child to be checked, the findChild function is called, which is shown in Fig. 7. For NodeCollapsed, first, the node is expanded to an ART node by passing it to the expandNode function (line 25). Next, the findChild function is recalled again, and the expanded node is passed to it (line 26).

```
findChild (node, key, depth)
 1 byte= key[depth]
 2 if node.type==Node4 // simple loop
 3   for (i=0; i<node.count; i=i+1)
 4     if node.key[i]==byte
 5       return node.child[i]
 6   return NULL
 7 if node.type==Node16 // SSE comparison
 8   key=_mm_set1_epi8(byte)
 9   cmp=_mm_cmpeq_epi8(key, node.key)
10   mask=(1<<node.count)-1
11   bitfield=_mm_movemask_epi8(cmp)&mask
12   if bitfield
13     return node.child[ctz(bitfield)]
14   else
15     return NULL
16 if node.type==Node48 // two array lookups
17   if node.childIndex[byte]!=EMPTY
18     return node.child[node.childIndex[byte]]
19   else
20     return NULL
21 if node.type==Node256 // one array lookup
22   return node.child[byte]
23 if node.type==NodeCollapsed
24   // expand node and recall
25   node = expandNode(node, key, depth)
26   return findChild(node, key, depth)
```

Fig. 7. findChild Algorithm.

The expandNode function is shown in Fig. 8. The main scenario is shown in lines 14 to 30. First, the collapsed node keys are partitioned into groups such that the ones that share the same depth key byte are stored in one group (line 14). Based on the generated groups count, the new expanded node type is chosen (lines 16-23). For each group, a new collapsed node is created, holding its keys, (line 26) and added to the new expanded node (line 29). A special case is that if the group key matches the queried key byte, the child collapsed node is expanded before adding it to the new node (lines 27, 28).

```
   expandNode (node, key, depth)
 1 if depth == treeLength //current is last level
 2   if node.keys.length <= 4
 3     newNode= makeNode4()
 4   else if node.keys.length <= 16
 5     newNode= makeNode16()
 6   else if node.keys.length <= 48
 7     newNode= makeNode48()
 8   else
 9     newNode= makeNode256()
10   for i to node.keys.length
11     leaf= makeLeaf(node.keys[i])
12     addChild(newNode, key[depth], leaf)
13 else
14   keyGroups= partitionKeys(node.keys, depth)
15   groupsCount= keyGroups.length
16   if groupsCount <= 4
17     newNode= makeNode4()
18   else if groupsCount <= 16
19     newNode= makeNode16()
20   else if groupsCount <= 48
21     newNode= makeNode48()
22   else
23     newNode= makeNode256()
24   for i to groupsCount
25     group= keyGroups[i]
26     child= makeNodeCollapsed(group.keys)
27     if group.key == key[depth]
28       child =expandNode (child, key, depth+1)
29     addChild(newNode, group.key, child)
30 return newNode
```

Fig. 8. expandNode Algorithm.

Another special case is represented by lines from 1 to 12. If the current level is the last level of the tree, there is no need to create additional child collapsed nodes. In this case, the new expanded node type is chosen based on the collapsed node keys count (lines 1-12), and each key is turned to a leaf node (line 11) and added directly to the expanded node (line 12).

In the expandNode function, some ART helper functions are used: makeNode*X* creates a new node of type *X*, makeLeaf creates leaf node, and addChild add a new child to an expanded node.

Insert: insert algorithm is, the same as that of the ART index, except that if the node where the new key will be inserted into is of type NodeCollapsed, the key is added directly into the node keys vector.

E. ART vs. IBART

As previously mentioned in the introduction section, the need for adaptive indexing techniques is raised for the systems of unpredictable workload and extremely large data sets. In such systems, there is no pattern of the incoming requests that can help in the decision of creating indexes before any query arrives. Furthermore, for big data systems, the full index creation process is a heavyweight process that requires plenty of time and resources. In addition, for such big systems, the probability that the incoming requests will access all the indexed values is very poor. So, creating an indexing holding all the related field values can be considered as a waste of time and resource. Therefore, adaptive indexing techniques, which IBART is one of them, are the optimal choice for those systems. For other systems, where there is a prior knowledge of how the data is queried and luxury of resources and idle time, ART, with no doubt, is the recommended strategy.

IV. EXPERIMENTS AND EVALUATION

In this section, we evaluate the proposed model by comparing its performance to the ARTful index. The experiments were done on an Intel Xeon E5 V3, 1.6 GHz CPU and a 64 GB RAM. We used Windows Server 2012 R2 in 64-bit mode as an operating system and GCC 5.3 as a compiler.

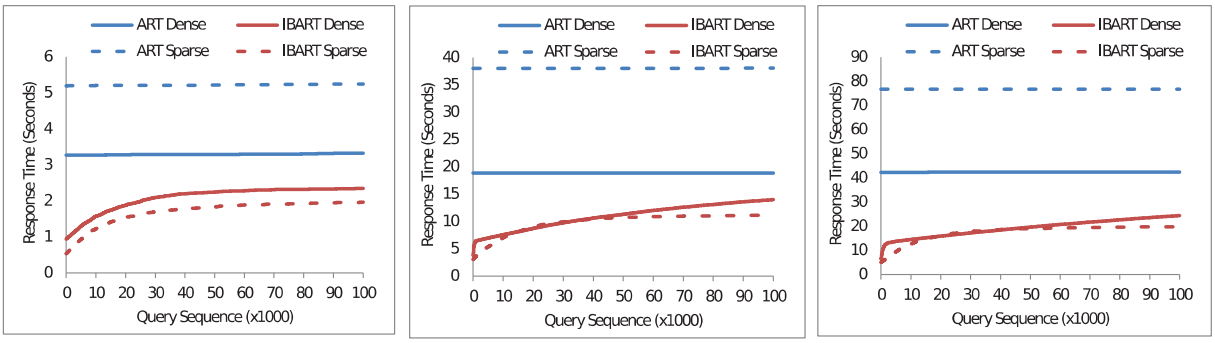
In the comparison, we used the source code provided by the ART authors [28]. The experiments compare the performance of ART and IBART assuming the index is created on runtime based on monitoring a dynamically changeable workload. We measured the cumulative response time of searching for a set of randomly chosen 100K keys. As the performance of the radix trees depends on the distribution of the indexed data, we repeated the experiments for a continuous generated key range from 1 to n and a randomly generated key set of size n, which represent dense and sparse data sets, respectively.

Fig. 9 shows the results of this experiment for indexes that contains 10M, 50M and 100M keys. The x-axis represents the performed lookups, and the y-axis represents the cumulative response time for each lookup. The cumulative time means that each point (x, y) represents the sum of the cost y for the first x queries.

From the results, we observe that at the beginning the ART takes some idle time to build its index tree. The idle time highly depends on the number of keys to be indexed. After creating the index, the cost of looking up any key is few micro seconds regardless the tree size. Another notice is that the idle time of indexing sparse data is almost twice that of indexing dense data.

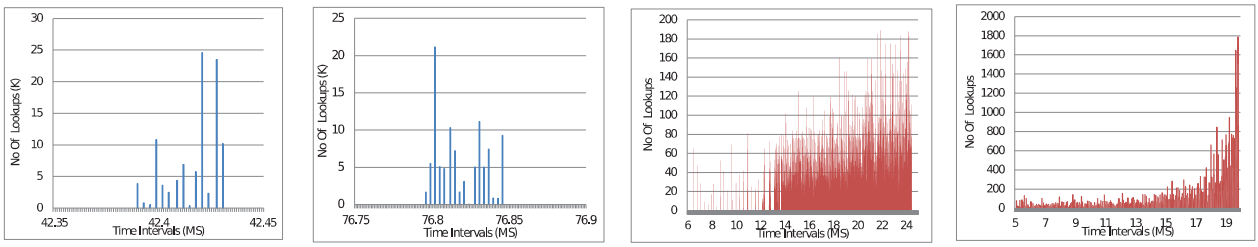
On the other hand, the IBART takes less idle time than ART to initiate its tree, not fully constructing it, and then it starts to respond to the first query. In the beginning, the curve grows fast. The reason is that almost all of the firstly incoming lookups need to expand some collapsed node of the tree, but as more queries are processed, the curve starts to grow at a lower rate as the subsequent incoming requests benefit from the branches expanded by the preceding ones.

An important observation is that the performance of IBART for sparse keys is better than that of dense keys in relatively small data set (10M) and it is almost the same for bigger sizes, unlike all other radix trees. The explanation of this behavior is that each expansion process depends on the number of keys sharing the branch to be expanded. As the keys distribution increases, the number of keys sharing the same branch decreases, so the cost of the branch expansion gets lower and, consequently, the performance increases.



(a) (b) (c)

Fig. 9 Cumulative response time in an index of (a) 10M, (b) 50M, and (c) 100M keys.



(a) (b) (c) (d)

Fig. 10 Histogram of query processing time intervals in an index of (a) dense ART, (b) sparse ART, (c) dense IBART, and sparse IBART.

Fig. 10 represents the histograms of query processing time intervals for 100M keys index. For IBART of dense and sparse keys, the idle time is 5 to 6 seconds, and then the rest of the 100K size workload is processed consecutively in the next 14 to 18 seconds.

As for ART of dense keys, the first query is penalized with an extra cost of 42 seconds before responding to it. After that, the rest of the workload is responded to in a few microseconds. The case for sparse keys is the same except that the idle time is 76 seconds.

From the histogram, we conclude that for both dense and sparse keys, IBART response for the whole workload before the ART starts responding to the first request.

V. Conclusions and Future Work

In this paper, we present the incremental branching adaptive radix tree (IBART), an adaptive indexing technique that takes the main concept of database cracking and applies it to the adaptive radix tree. The IBART implies the lightweight, dynamic, and adaptive nature of database cracking and the space-efficiency and agility of ART along with all its other features like path compression, lazy expansion, and the ability to store all common data types. The performance of IBART is compared to that of ART for a dynamically created index at runtime. The experiments show that IBART is much faster for systems of dynamic nature where the workload has no predefined pattern.

As discussed in the paper, the expansion process depends highly on the number of keys sharing the branch to be expanded. This means that while branching, the IBART loses the key feature of radix tree, which is that its performance is not a factor of the number of indexed elements. So in the future, we aspire to find a branching technique that does not rely on the indexed keys count. One idea is to use a hash table as an inner architecture of the collapsed nodes.

Another aspect that we intend to work on in the future is to parallelize the tree initialization and the collapsed node expansion process by using multiple unsynchronized threads in order to make the best use of the modern CPUs.

References

[1] N. Bruno, S. Chaudhuri, A. C. König, V. Narasayya, R. Ramamurthy, and M. Syamala, "AutoAdmin Project at Microsoft Research: Lessons Learned," *Bull. IEEE Comput. Soc. Tech. Comm. Data Eng.*, pp. 1–8, 2011.

[2] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback, "Self-tuning database technology and information services: from wishful thinking to viable engineering," *Proc. 28th Int. Conf. Very Large Data Bases*, pp. 20–31, 2002.

[3] G. Graefe and H. Kuno, "Self-selecting, self-tuning, incrementally optimized indexes," *Proc. 13th Int. Conf. Extending Database Technol. - EDBT '10*, p. 371, 2010.

[4] S. F. Rodd and U. P. Kulkarni, "Adaptive self-tuning techniques for performance tuning of database systems: a fuzzy-based approach with tuning moderation," *Soft Comput.*, vol. 19, no. 7, pp. 2039–2045, 2015.

[5] S. Chaudhuri and V. R. Narasayya, "Self-tuning database systems: a decade of progress," *Proc. 33rd Int. Conf. Very large data bases*, pp. 3–14, 2007.

[6] S. Idreos, S. Manegold, and G. Graefe, "Adaptive indexing in modern database kernels," in *Proceedings of the 15th International Conference on Extending Database Technology - EDBT '12*, 2012, pp. 566–569.

[7] S. Idreos, M. Kersten, and S. Manegold, “Database Cracking,” *CIDR ’07 3rd Bienn. Conf. Innov. Data Syst. Res.*, pp. 68–78, 2007.

[8] F. M. Schuhknecht, A. Jindal, and J. Dittrich, “An experimental evaluation and analysis of database cracking,” *VLDB J.*, 2015.

[9] V. Leis, A. Kemper, and T. Neumann, “The adaptive radix tree: ARTful indexing for main-memory databases,” in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 2013, pp. 38–49.

[10] A. Gupta and I. S. Mumick, *Materialized views: techniques, implementations, and applications*. MIT Press Cambridge, MA, USA ©1999, 1999.

[11] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou, “Vertical partitioning algorithms for database design,” *ACM Trans. Database Syst.*, vol. 9, no. 4, pp. 680–710, 1984.

[12] S. Ceri, M. Negri, and G. Pelagatti, “Horizontal data partitioning in database design,” in *Proceedings of the 1982 ACM SIGMOD international conference on Management of data - SIGMOD ’82*, 1982, p. 128.

[13] Y. Manolopoulos, Y. Theodoridis, and V. J. Tsotras, *Advanced Database Indexing*, vol. 17. Boston, MA: Springer US, 2000.

[14] A. Gani, A. Siddiqa, S. Shamshirband, and F. Hanum, “A survey on indexing techniques for big data: taxonomy and performance evaluation,” *Knowl. Inf. Syst.*, 2015.

[15] S. Idreos, “Database Cracking: Towards Auto-tuning Database Kernels,” van Amsterdam, 2010.

[16] S. Idreos, M. L. Kersten, and S. Manegold, “Self-organizing Tuple Reconstruction in Column-stores,” *Sigmod*, pp. 297–308, 2009.

[17] G. Graefe and H. Kuno, “Self-selecting, self-tuning, incrementally optimized indexes,” in *Proceedings of the 13th International Conference on Extending Database Technology - EDBT ’10*, 2010, p. 371.

[18] S. Idreos, S. Manegold, H. Kuno, and G. Graefe, “Merging what’s cracked, cracking what’s merged,” *Proc. VLDB Endow.*, vol. 4, no. 9, pp. 586–597, Jun. 2011.

[19] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap, “Stochastic database cracking,” *Proc. VLDB Endow.*, vol. 5, no. 6, pp. 502–513, Feb. 2012.

[20] V. Alvarez, F. M. Schuhknecht, J. Dittrich, and S. Richter, “Main memory adaptive indexing for multi-core systems,” in *Proceedings of the Tenth International Workshop on Data Management on New Hardware - DaMoN ’14*, 2014, pp. 1–10.

[21] S. Richter, J.-A. Quiané-Ruiz, S. Schuh, and J. Dittrich, “Towards zero-overhead static and adaptive indexing in Hadoop,” *VLDB J.*, vol. 23, no. 3, pp. 469–494, Jun. 2014.

[22] E. Petraki, S. Idreos, and S. Manegold, “Holistic Indexing in Main-memory Column-stores,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD ’15*, 2015, pp. 1153–1166.

[23] P. Holanda, C. Wiskunde, and P. Holanda, “SPST-Index : A Self-Pruning Splay Tree Index for Caching Database Cracking SPST-Index : A Self-Pruning Splay Tree Index for Caching Database Cracking,” no. March, 2017.

[24] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki, “Slalom : Coasting Through Raw Data via Adaptive Partitioning and Indexing,” *Pvldb*, vol. 10, no. 10, pp. 1106–1117, 2017.

[25] J. Rao and K. A. Ross, “Making B + -Trees Cache Conscious in Main Memory,” pp. 475–486, 2000.

[26] M. Boehm, B. Schlegel, P. B. Volk, U. Fischer, D. Habich, and W. Lehner, “Efficient In-Memory Indexing with Generalized Prefix Trees,” no. 2, pp. 227–246.

[27] C. Kim *et al.*, “FAST : Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs,” pp. 339–350.

[28] T. N. Viktor Leis, Alfons Kemper, “Adaptive Radix Tree source code.” [Online]. Available: https://db.in.tum.de/~leis/index/ART.tgz.