The DCB-Tree: A Space-Efficient Delta Coded Cache Conscious B-Tree

Robert Binna(✉), Dominic Pacher, Thomas Meindl, and Günther Specht

Databases and Information Systems, Institute of Computer Science,
University of Innsbruck, Innsbruck, Austria
{robert.binna,dominic.pacher,thomas.meindl,gunther.specht}@uibk.ac.at

Abstract. Main-memory index structures have become mainstream for a large number of problem domains. However, in the case of web-based datasets, which feature exponential growth, it is an ongoing challenge to fit those data entirely in main-memory. In this paper, we present the DCB-Tree, an extremely space efficient main-memory index structure for the storage of short fixed-size keys. It features a two-stage cache-line aligned node layout. In comparison to other main-memory index structures it reduces the amount of memory required by 80 % in the best and by 30 % in the worst case. Although it is tailored towards space consumption, it features good overall performance characteristics. In particular, in the case of very large real world datasets it provides performance equal or superior to state of the art main-memory index structures.

Keywords: Indexing · Main-memory · Delta-encoding · Cache-optimized

1 Introduction

With the amount of main-memory capacities increasing, many databases can be kept entirely in main-memory. However, this is not the case for all domains. For example, the exponential growth of web-based datasets results in huge semantic web data or full-text corpuses. Although data can be sharded and distributed onto several hosts, it is still desirable to reduce the amount of memory needed and therefore to reduce the monetary cost as well. In the case of RDF-Stores [5,18] a common approach to reduce the amount of memory needed for indexing such kind of data, is to solely store fixed size keys instead of long variable length values and place the original value in a dictionary. Hence to reduce the amount of memory required, the dictionary or the fixed length keys can be compressed.

As the main purpose of index structures is to improve the search and query performance, the space consumption of such indexes is a trade-off between access performance and compression ratio. Nevertheless, the compression overhead can be mitigated due to the memory wall effect [22], which states that the improvement in microprocessor speed exceeds the improvement in DRAM performance. Currently DRAM speed is more than two orders of magnitude slower than CPU speed. This situation is comparable to the performance disparity between main-memory and disks. As disk based index structures use compression techniques

A. Jagatheesan et al. (Eds.): IMDM 2013/2014, LNCS 8921, pp. 126–138, 2015.
DOI: 10.1007/978-3-319-13960-9_10

to reduce the required number of disk seeks [8]. According to the famous quote "memory is the new disk" [10] it can be inferred that "cache is the new RAM". Hence, compressing data on a cache-line level can trade instructions required for compression against CPU cycles gained by reducing the number of cache misses. The memory wall effect becomes even more significant for index structures consisting of several 100 millions or even billions of elements as the number of elements residing in CPU caches is limited. Therefore, the performance is mainly bound by the overall number of memory transfers required.

Hence, we identify three requirements for main-memory index structures, particularly in the case of large real world datasets. First, the data structure has to be space efficient. Second, it has to consider cache hierarchies of contemporary CPUs. Third, it has to support incremental updates as read-only indexes are only appropriate for a limited number of use cases.

Currently two categories of main-memory indexes, which address some of these issues, can be identified. On the one hand, read-only index structures like FAST [13] provide cache-conscious search operations as well as decent space utilization, but do not support update operations. On the other hand, main-memory index structures like ART [15] or the CSB^+-Tree [20] provide cache-conscious search as well as update operations, but provide only limited capabilities in terms of space utilization. However, to the best of our knowledge no method satisfying all three requirements exists, which provides space efficiency, cache-conscious design and update functionality.

Therefore, we present the Delta Cache Conscious B-Tree (DCB-Tree) combining cache-conscious operations with efficient space utilization. It is a main-memory B-Tree [3] based index structure tailored for the storage of short fixed-size keys. A hierarchical cache-line aligned node layout reduces the number of cache-misses and the delta encoding of keys along this hierarchical layout reduces the amount of space needed for each entry.

Hence, our contributions in this paper are:

- The main-memory Delta Cache Conscious B-Tree (*DCB-Tree*)-Tree. It is a space efficient, cache-conscious index for the storage of short fixed size keys.
- Novel algorithms for lookup and insertion, which are specific for the *DCB-Tree*.
- An evaluation of the *DCB-Tree*, which shows that the *DCB-Tree* provides superior space utilization, while providing equal or better performance for large real world datasets.

The remainder of this paper is structured as follows. Section 2 discusses the related work. Section 3 introduces the *DCB-Tree*, its structure and the algorithms used. Section 4 presents the evaluation and its results. Section 5 draws a conclusion and gives a short overview about future work.

2 Related Work

The work presented in this paper is related to the area of index structures and index compression techniques.

In the area of index structures allowing sequential access, comparison-based tree structures as well as trie-based index structures prevailing. The B-Tree [3,8] represents the dominating disk based index structure and is the basis of many relational database systems. While variations of binary search trees like the Red Black Tree [2,11] or the T-Tree [14] were the dominating main-memory index structures until the 1990's, Rao *et al.* [19] showed that B-Tree based structures exceed binary tree index structures due to their better cache-line utilization. Rao *et al.* further presented the CSB+ tree [20], which optimizes cache-line utilization by reducing the number of pointers through offset calculation in the index part. While the CSB+ tree improves memory utilization, no key or pointer compression was applied. Due to the fact that cache optimized trees provide only limited performance when used as disc based structures, Chen *et al.* presented the Fractal Prefetching B^+-Tree [7], which facilitates a hierarchical cache-optimized layout optimizing disc as well as main memory performance.

Another approach aiming for a cacheline optimization is the BW-Tree [16] by Levandoski *et al.* which is optimized for high concurrency by facilitating atomic compare and swap operations instead of locks. A further direction of research is to facilitate the data-parallel features of modern hardware to improve the search operation within the tree nodes. The Fast Architecture Sensitive Tree (FAST) by Kim *et al.* [13] and the k-array search based approach by Schlegel *et al.* [21] use SIMD operations to speed up search operations. While it was shown that both trees provide improved search performance, they were designed as read-only index lacking update operations. Another approach to speed up the search operation is to use trie-based index structures [9]. However, a major drawback of tries is the worst-case space consumption. The Adaptive Radix Tree (ART) [15] by Leis *et al.* represents a trie variation dedicated to modern hardware, which mitigates this worst-case space consumption by using adaptive node sizes as well as a hybrid approach for path compression. Moreover, the authors showed that the ART tree is able to outperform FAST and under certain conditions also hashtables. Another approach based on tries is the Masstree [17] by Mao *et al.*, which is a trie with fanout 2^{64} where each trie node is represented by a B^+-Tree storing an 8 byte portion of the key. This design results in good performance for long shared prefixes.

In the domain of index compression techniques several different approaches to compress the index part as well as the file part of B-Trees were developed. The reason is that the performance of index structures is heavily bound by the branching factor. A common compression scheme related to the compression of the index part is prefix or rear compression. The prefix B-Tree [4] by Bayer and Unterauer uses prefix compression on a bit level to only store partial keys in the index part of the tree. Furthermore, they soften the B-Tree properties to select partial keys with the shortest length. Bohannon *et al.* extended the concept of partial keys in their pkT-trees and pkB-trees [6] to improve cache and search performance. In the index part they use fixed size portions of the prefix to optimistically compare with the search key. If the comparison cannot be performed, a pointer to the full index key is dereferenced. The authors point out that the partial key is superior in terms of performance for larger keys only. While this scheme improves the cache-line utilization, it imposes a memory overhead due to the overhead of the pointer as well as the partial key itself. A more recent approach for using partial keys was incorporated in FAST [13]. It compresses the index keys by applying a SIMD based approach to only store those bits that actually differ.

3 DCB-Tree

In this section, we present the Delta Cache Conscious B-Tree (*DCB-Tree*). The major goal of the DCB-Tree is to store fixed size keys of sizes up to 8 bytes in the *DCB8-Tree* and keys of up to 16 bytes in the larger *DCB16-Tree*. Further aims of the *DCB-Tree* are a low memory footprint, update ability and taking cache hierarchies into account to provide decent performance. The two variations *DCB8-Tree* and the *DCB16-Tree* differ only in the underlying integer type. The reason that two versions exist, is that the 8 byte integer type is mapped to a native data type and therefore provides better performance compared to the custom 16 byte integer type. In the following, we will use the term *DCB-Tree* synonymously for both the *DCB8-Tree* and the *DCB16-Tree*.

The intended use of the *DCB-Tree* is to provide a clustered in-memory index for short fixed-length keys, which occur in triple stores or inverted indexes. Due to the dataset sizes in those areas, the *DCB-Tree* focuses primarily on the reduction of the overall memory footprint to ensure that datasets can be processed in main-memory only. To reduce the memory footprint, the *DCB-Tree* exhibits the circumstance that generally keys are not randomly distributed. Moreover, in real world scenarios coherence can be found in the datasets. This circumstance is utilized by the *DCB-Tree* for the encoding of keys and for the encoding of pointers. The *DCB-Tree* is an n-ary search tree incorporating ideas of B-Trees [3], B^+-Trees [8] and CSB^+-Trees [20]. Values are only stored at the leaf-level. Due to the fact that the *DCB-Tree* is designed as a clustered index structure, no pointers to the actual record are stored on the leaf node level. In the following we describe the node and bucket layout, discuss the pointer encoding and memory layout and explain the algorithms for lookup and insertion.

3.1 Two Stage Node Layout

Each node, index node as well as leaf node, has a two stage layout. The first stage is the header section and the second stage consists of the buckets containing the content (keys). Furthermore, each node has an implicit offset which is defined by its corresponding parent node entry. In the case of the root node, the offset is defined to be zero. An example node layout is illustrated in Fig. 1.

The header section of each node contains header-entries (H_1-H_n), which are uncompressed keys used as separators between buckets. For instance, header-entry H_1 is larger than any key (K_i) in $Bucket_1$ but smaller or equal to the smallest key in $Bucket_2$. In this way the header-entries can be used to determine

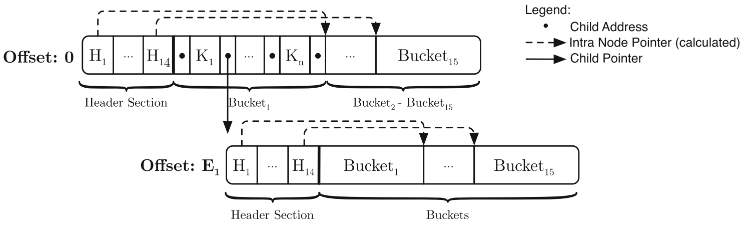


Fig. 1. DCB-tree layout overview

the address of the corresponding bucket address. In Fig. 1 this relationship is visualized by the intra node pointers. However, these intra node pointers are not stored but are calculated relative to the node's base address. Hence, for nodes with header sections containing at most n header-entries at most $n + 1$ buckets are supported. In the case of the DCB-Tree the size of the header section as well as the size of each bucket is determined to two cache-lines. The reason is that the hardware prefetchers[1] of modern CPUs automatically fetch the adjacent cache-line in case of a cache miss. As the number of bytes required per header entry is equal to the maximum key length, headers in a *DCB16*-Tree are twice as large as in a *DCB8-Tree*. Therefore, to be able to address the same number of buckets as in a *DCB8-Tree*'s node the header section of a *DCB16*-Tree spans four cache-lines. As the header-entries are used to address the corresponding buckets, no more than 16 content buckets can be used without increasing the header section size. To ensure that each node is page aligned and that no TLB-miss occurs during the processing of a single node, the node size is fixed to 2kiB. Furthermore the 2kiB node size is tailored to the address translation covered in Sect. 3.3. This node design ensures that at most two cache misses occur on each tree level of a *DCB8-Tree*. In a *DCB16-Tree* this depends on the prefetch policy of the CPU, but tends to three cache misses.

3.2 Bucket Structure

As described in the previous section, the content of each node is stored in its buckets. To distinguish buckets located in index nodes from buckets located in leaf nodes, we denote buckets in index nodes as index buckets and buckets in leaf nodes as leaf buckets. Due to the fact that the bucket structure is similar to the node structure in B^+-Trees we only discuss the properties which are different from the B^+-Tree's node structure.

Index buckets contain keys and pointers to the corresponding subtree. Each index bucket contains a header section and a content section. The header section stores the number of entries in the content section and the encoding information for keys and pointers. The encoding information for pointers consists of the number of bytes (*Pointer Bytes*) used to store the largest pointer in the bucket.

[1] http://tinyurl.com/on8ccx3.

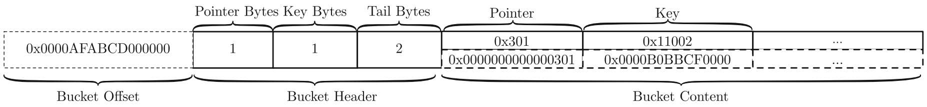


Fig. 2. Index bucket entry

Keys in index buckets are generated in the course of leaf node splits, such that the lower bytes tend to be zero. Therefore a tail compression is applied on the bucket's keys and the minimum number of tail zero bytes (*Tail Bytes*) is stored in the bucket's header section. Furthermore, the keys are delta coded relative to the bucket's offset and placed in its content section. The maximum number of bytes, which is required to store such an encoded keys, is put into the bucket's header section (*Key Bytes*). An example illustrating pointer as well as key encoding for an index bucket can be seen in Fig. 2. In this figure solid boxes represent stored values, while dashed boxes represent calculated values. For instance, the bucket offset is inferred during tree traversal. Dashed boxes below pointers or keys contain their decoded representations.

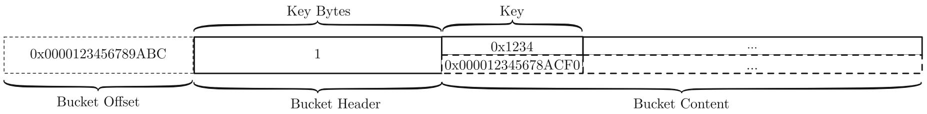


Fig. 3. Leaf bucket entry

Leaf buckets store keys only. Hence their section contains the number of keys in the content section and the information used to encode them. As keys contained within a leaf bucket are highly coherent, the same fixed-length delta based encoding as used for index buckets can be applied. Therefore, the encoding information consists of the maximum number of bytes needed to encode a single key. This is equal to the number of bytes required to encode the leaf bucket's largest key. An example which illustrates the encoding of a leaf bucket can be seen in Fig. 3. The semantics of the dashed and solid boxes is analogous to Fig. 2.

3.3 Pointer Encoding and Memory Layout

It has previously been shown [19] that a large portion of space within index nodes is dedicated to pointer information. Since 64-bit architectures have become mainstream, the space dedicated to pointers has an even higher impact. Therefore, we try to reduce the amount of space dedicated to pointer information with a twofold strategy. On the one hand, the number of pointers required is reduced. This is done by designing the data structure as a clustered index for short fixed sized keys, which eliminates the need for pointers at the leaf node level. Furthermore, by facilitating the nested node layout described in Sect. 3.1, intra-node pointers are eliminated. This approach is similar to the concept of node groups in CSB^{+}-Trees. On the other hand, the space occupied by each pointer is reduced. This is achieved by using node IDs instead of absolute pointers. Even in the case of huge datasets spanning hundreds of billion of entries, 4 bytes for addressing a single node is sufficient. Therefore, a custom memory allocator is used, which allocates fixed node size chunks from a pool of large buffers and is able to translate node IDs to absolute addresses. The overhead of calculating the absolute node address is negligible, as 2^n sized buffers are used.

3.4 Algorithms

As a *DCB-Trees* resembles a special version of a B-Tree, the basic algorithms for insertion and update are the same. The huge difference in comparison to standard B-Trees is that each node features a two stage layout, with key and pointer compression on a per bucket basis. We therefore describe only the parts of the algorithms and operations that are different compared to B-Trees and B^+-Trees [3,8]. Moreover, only insertion as well as lookup operations are considered. Nonetheless, algorithms for delete operations can be inferred analogously.

Lookup. In the case of a lookup operation, the two stage node layout results in the following three steps, which are needed for searching a node.

1. A linear search is executed on the node's header to determine the corresponding bucket and its offset. The bucket offset of the first bucket is equal to the node offset. In any other case, the bucket offset is equal to the largest header entry which is smaller or equal to the search key.
2. The search key is encoded. Therefore, the bucket offset is subtracted from the search key. In the case of an index bucket, the search key is tail encoded to match the encoding of the bucket to search.
3. A lookup operation is executed on the bucket using the encoded search key. In the case of an index bucket search, the key preceding the matching pointer becomes the node offset in the next search step.

Insert. To insert a key into a *DCB-Tree*, first the target leaf bucket is determined by the lookup operation explained in the previous paragraph. Second, it is determined whether sufficient space is available to insert the key and whether a potential recode operation succeeds. Such a recode operation is required if the new key is larger than any existing entry in the bucket. If both conditions can be satisfied, the bucket is encoded and the key inserted. If one of the conditions fail, the following overflow handling strategies are applied in the given order.

1. In the case of inserting keys in ascending order, buckets tend to be only half full. To prevent this situation, keys are rotated to the left sibling bucket, if sufficient space is available. This corresponds to a local rotation.
2. Otherwise a bucket split becomes necessary. Unless the node is full, the bucket split is executed and the minimum key of the right bucket is inserted as a new separator in the node's header section.
3. If a bucket split cannot be performed and the left sibling node can contain further buckets, buckets are locally rotated to the left sibling node and the

insertion is retried. In the case of a bucket rotation the node offset must be recalculated, which can lead to another bucket split. Furthermore, the new node offset must be propagated to the parent node.

4. If no bucket rotation can be applied a node split is required. The buckets are distributed, the node offset for the right node is calculated and the pointer to the right node together with its offset is inserted into the parent node.

For the generation of the node offset, in case of a leaf-node split, Algorithm 1 is used. It calculates a separator between the left and the right node with as many trailing zero bytes as possible. Moreover, it does not calculate the shortest common prefix between the largest left and the lowest right value, as in case of the prefix B-Tree [4], but the shortest common prefix between the mean value of both values and the lowest right value. The reason is that it tries to balance the size of the right node's first bucket values while still providing a decent prefix.

Algorithm 1. Tail Compressible Mean

```
procedure TAILCOMPRESSIBLEMEAN($lower, upper$)
    $mean \leftarrow (upper + lower)/2$
    $upperMask \leftarrow -1l \ggg numberOfLeadingZeros(upper)$
    $tailCompressableBits = log_2((mean \oplus upper) \wedge upperMask)$
    return $(-1l \ll tailCompressableBits) \wedge upper$
end procedure
```

4 Evaluation

In this section we evaluate the *DCB-Tree*. Therefore we conduct two benchmarks. The first compares the memory consumptions with other main-memory data structures. The second evaluates the runtime performance.

4.1 Benchmark Environment

All benchmarks are executed on the Java Runtime Environment version 1.8.0_05 with the following system properties set: `-XX:NewRatio=3 -Xmx90g -Xms40g -XX:+UseConcMarkSweepGC -XX:MaxDirectMemorySize=90g`.

For the evaluation we used a server with an Intel Xeon L5520 running at a clock speed of 2.27 GHz clock speed, 64 KB L1 cache per core, 256 KB L2 cache per core and 8MB L3 shared cache. The server has 96 GB of DDR3/1066 RAM and runs CentOS 6.5 with Linux Kernel 2.6.32.

4.2 Evaluated Data Structures

As the *DCB-Tree* is implemented in Java, all benchmarks are evaluated on the Java Platform. The implementation of the DCB-Tree is available online[2]. We

[2] http://dbis-informatik.uibk.ac.at/static/ma/robert/imdm/imdm2014.zip.

evaluated the two variations *DCB8-Tree* and *DCB16-Tree*. Due to the lack of built-in 16 byte wide integers a custom integer data type is used as the underlying data type for the *DCB16-Tree*.

As contestants, the TreeSet[3] representing the Java Platform's standard implementation of a Red-Black Tree [11] and a port of the ART Tree [15] were used. As the ART tree is originally available as C++ implementation[4] we created a port for the Java Language, which is available online. Due to the lack of SIMD operations on the Java Platform, the lookup in the nodes containing up to 16 keys of the ART Tree had to be implemented by linear search. Although it is expected that the ART port is slower than the C++ implementation due to the overhead incurred by the Java Virtual Machine, the same overhead is applied to all contestants. For keys up to the length of 8 bytes the key is encoded inside the pointer as it is the case in the original implementation. The reason is that 8 bytes are already reserved for the pointer. In the case of keys larger than 8 bytes the ART Tree is used as a secondary index structure as the 16 byte key cannot be encoded in an 8 byte pointer.

It is important to note, that in the case of the ART Tree as well as the DCB-Tree, pointers represent relative offsets in a direct ByteBuffer[5]. This is similar to an offset for an array in C.

4.3 Datasets

In the scenario of keys up to 8 bytes length we use three different datasets. The first dataset contains dense values ranging from 0 to n. The second dataset contains random values. Finally, two real world dataset are used. On the one hand side triples of the Yago2 dataset [12] are encoded as 8 byte sized keys in the following way: The lowest 26 bits are used for the object id. Bits 27 to 37 store the predicate information and the bits ranging from 38 to 63 are used for the subject information. On the other hand triples from the DBpedia [1] dataset version 3.9 are encoded as 16 byte sized key. Each triple is encoded in a single 16 byte integer, such that the lowest 4 bytes represent the object id, the next 4 bytes the predicate id and the next 4 bytes the subject id.

For the sake of simplicity we subsequently denote these four datasets as *Dense*, *Random*, *Yago* and *DBpedia*.

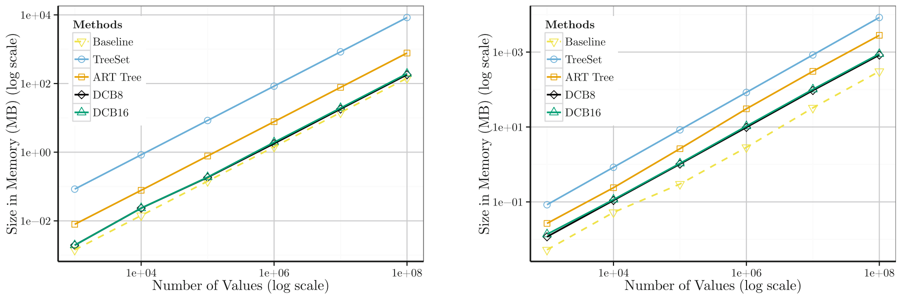
4.4 Memory Consumption

To evaluate the memory consumption we insert 10 K, 100 K, 1 M, 10 M and 100 M keys of each dataset into the index structures and measure the space consumption for each structure. For *DBpedia*, we use dataset sizes ranging from 100 K up to 1 B keys. The space consumption is summarized in Table 1, with the best values written in bold. This table presents the bytes used per entry

[3] http://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html.

[4] http://www-db.in.tum.de/~leis/index/ART.tgz?lang=de.

[5] http://docs.oracle.com/javase/8/docs/api/java/nio/ByteBuffer.html.



(a) Dense Dataset

(b) Yago Dataset

Fig. 4. Memory consumption

for 100 million randomly inserted keys of each data set. Moreover the *Baseline* value used in Table 1 represents the bytes needed per key, given all keys are delta coded relative to their direct predecessor and encoded with a byte-level fixed-prefix encoding. Furthermore the memory consumption is exemplarily visualized for *Dense* in Fig. 4a and for *Yago* in Fig. 4b.

As it can be seen in both Figures, as well as in Table 1, the overhead in terms of memory consumption between *DCB8* and *DCB16* is negligible and can be explained by the additional cache-lines used in the header section of *DCB16*. Hence, in the rest of this subsection we will use *DCB* synonymously for *DCB8* and *DCB16*.

The results of all experiments show that the *DCB*-Tree has the best space utilization of the evaluated data structures. In the best case (*Dense*) it uses 30 % more space than the Baseline. Even in the worst case it uses only three times more space compared to the Baseline. The DCB-Tree uses between two third of the memory of the second best index structure (ART-Tree) in the case of *Random*, and up to five times less space in the case of *DBpedia*. In the case of *DBpedia* it has to be considered that the keys cannot be stored inside ART. Therefore 16 bytes of the 60 bytes per entry are dedicated to the storage of the keys itself. Considering only the space required for the index part, the *DCB*-Tree uses only one third of the space. It can be seen in Table 1 that this ratio is equal for *Yago*, which represents a scale-free network as well. In all experiments it can be seen that the TreeSet performs worst. For each dataset it consumes about an

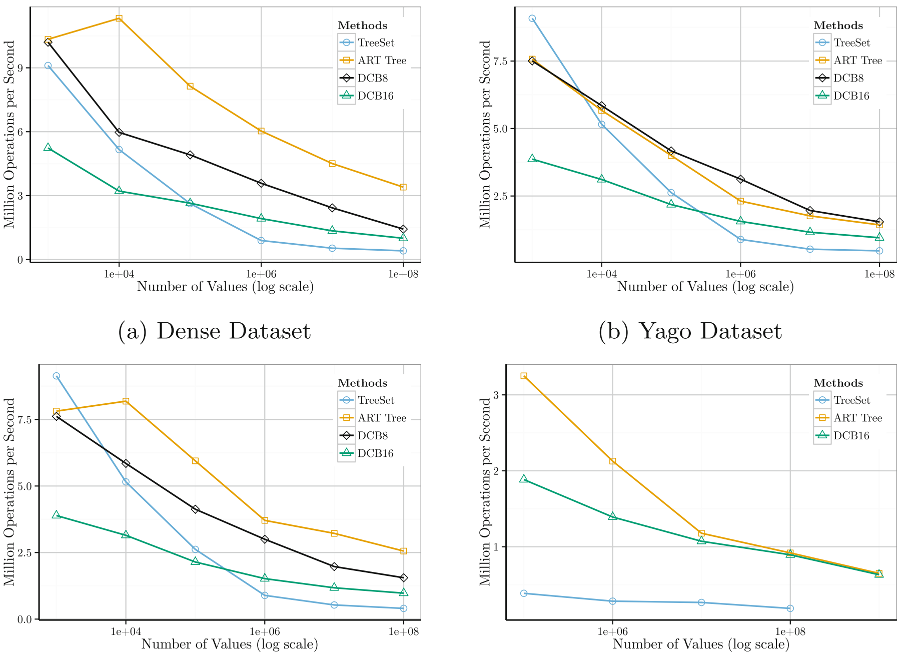
Table 1. Memory consumption per key for TreeSet, ArtTree and DCB in datasets of 100 million values

Dataset	Baseline	TreeSet	δ_{Tree}	ArtTree	δ_{Art}	DCB8	δ_{DCB8}	DCB16	δ_{DCB16}
Dense	1.5	88	58.67	8.1	5.4	1.89	**1.26**	2.03	1.35
Yago	3.2	88	27.5	29.4	9.19	8.73	**2.73**	9.36	2.92
Randomt	5.45	88	16.15	18.67	3.43	11.23	**2.06**	12.04	2.21
DBPedia	3.78	96	25.4	60.01	15.88	NA	NA	12.66	**3.35**

order of magnitude more space per key than the *DCB*-Tree. The reason is that it uses no compression and has a poor payload-to-pointer ratio.

4.5 Runtime Performance

To evaluate the runtime behavior, the same datasets and sizes as in the memory benchmark are used. For each configuration, 100,000,000 lookup operations are issued in a random order and the number of operations performed per second are measured. The results for *Dense* is shown in Fig. 5a, for *Random* in Fig. 5c, for *Yago* in Fig. 5b and for *DBpedia* in Fig. 5d. It can be observed that regarding the artificial datasets *Dense* and *Random*, the ART Tree processes about 50 % more lookups per second than the *DCB8* tree. Although the Red Black Tree has very good performance for datasets having less than 10,000 entries, for larger datasets the performance drops significantly and is surpassed by both the *DCB*-Tree as well as the ART-Tree.



(a) Dense Dataset

(b) Yago Dataset

(c) Random Dataset

(d) DBpedia Dataset

Fig. 5. Lookup performance

In case of the large real world datasets *Yago* and *DBpedia* the DCB-Tree is on the same level or superior to the ART Tree. For *Yago*, depending on the dataset size, DCB8 processes between 7 % and 30 % more operations per second than ART. For *DBpedia*, *DCB-16* has an equivalent runtime performance as ART for more than 10,000,000 entries. The reason that *DCB-16* performs up to 50 % worse than ART for dataset sizes smaller than 10,000,000, is the overhead of the custom 16 byte integer implementation, as no native 16 byte integer datatype is available on the Java Platform. ART is not affected by this, because it performs a byte wise comparison. Nevertheless, due to its tree height and the increased number of cache misses, the performance of ART drops significantly for datasets larger than 10,000,000 entries.

The reason that the TreeSet is only evaluated for dataset sizes of up to 100,000,000 entries is that the amount of memory required exceeds the amount of RAM available in our benchmark environment (more than 96 GB).

5 Conclusion and Future Work

In this paper we presented the DCB-Tree, a cache-conscious index structure for the storage of short fixed size keys. The DCB-Tree combines a hierarchical cache aligned node layout with delta encoding and pointer compression. The evaluation results show the best memory utilization among the contestants, while providing equal or better performance for large real world datasets.

We presented algorithms for insertion and search operations and described the influence of the two-stage node layout on B-Tree operations. Furthermore, the DCB-Tree was evaluated against two other index structures, namely the ART Tree and a Red Black Tree on artificial as well as on real world datasets. We show that for dense as well as for large real world dataset the DCB-Tree requires only 20 % of memory compared to other state of the art index structures. Moreover, our evaluation shows that the DCB-Tree provides decent performance using artificial datasets. In the case of large real world datasets it is equivalent or superior to state of the art in-memory index structure ART, while providing a more efficient space consumption. In future work, we will investigate other encoding strategies to further reduce the amount of memory required. Furthermore, we plan to integrate the DCB-Tree into RDF-Stores as well as to use it as a basis for full text indexes.

References

1. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: DBpedia: a nucleus for a web of open data. In: Aberer, K., et al. (eds.) ISWC/ASWC 2007. LNCS, vol. 4825, pp. 722–735. Springer, Heidelberg (2007)
2. Bayer, R.: Symmetric binary B-Trees: data structure and maintenance algorithms. Acta Informatica **1**(4), 290–306 (1972)
3. Bayer, R., McCreight, E.: Organization and maintenance of large ordered indices. In: Proceedings of the SIGFIDET (now SIGMOD) 1970, p. 107. ACM Press, New York (1970)
4. Bayer, R., Unterauer, K.: Prefix B-Trees. ACM Trans. Database Syst. **2**(1), 11–26 (1977)

5. Binna, R., Gassler, W., Zangerle, E., Pacher, D., Specht, G.: SpiderStore: exploiting main memory for efficient RDF graph representation and fast querying. In: Proceedings of Workshop on Semantic Data Management (SemData) at VLDB (2010)
6. Bohannon, P., Mcllroy, P., Rastogi, R.: Main-memory index structures with fixed-size partial keys. In: Proceedings of SIGMOD 2001, vol. 30, pp. 163–174. ACM Press, New York, June 2001
7. Chen, S., Gibbons, P.B., Mowry, T.C., Valentin, G.: Fractal prefetching B+-Trees. In: Proceedings of SIGMOD 2002, p. 157. ACM Press, New York (2002)
8. Comer, D.: Ubiquitous B-Tree. ACM Comput. Surv. **11**(2), 121–137 (1979)
9. Fredkin, E.: Trie memory. Commun. ACM **3**(9), 490–499 (1960)
10. Gray, J.: Tape is dead, disk is tape, flash is disk, RAM locality is king, Gong Show Presentation at CIDR (2007)
11. Guibas, L.J., Sedgewick, R.: A dichromatic framework for balanced trees. In: 19th Annual Symposium on Foundations of Computer Science (SCFS 1978), pp. 8–21. IEEE, October 1978
12. Hoffart, J., Suchanek, F.M., Berberich, K., Lewis-Kelham, E., de Melo, G., Weikum, G.: YAGO2: exploring and querying world knowledge in time, space, context, and many languages. In: Proceedings of WWW 2011, p. 229. ACM Press, New York (2011)
13. Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A.D., Kaldewey, T., Lee, V.W., Brandt, S.A., Dubey, P.: FAST: fast architecture sensitive tree search on Modern CPUs and GPUs. In: Proceedings of SIGMOD 2010, p. 339 (2010)
14. Lehman, T.J., Careay, M.J.: A study of index structures for main memory database management systems. In: Proceedings of VLDB 1986, pp. 294–303 (1986)
15. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: ARTful indexing for main-memory databases. In: Proceedings of ICDE 2013, pp. 38–49. IEEE, April 2013
16. Levandoski, J.J., Lomet, D.B., Sengupta, S.: The Bw-tree: A B-tree for new hardware platforms. In: Proceedings of ICDE 2013, pp. 302–313 (2013)
17. Mao, Y., Kohler, E., Morris, R.T.: Cache craftiness for fast multicore key-value storage. In: Proceedings of the 7th ACM European Conference on Computer Systems - EuroSys 2012, p. 183 (2012)
18. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. In: Proceedings of VLDB Endowment, vol. 1, pp. 647–659, August 2008
19. Rao, J., Ross, K.A.: Cache Conscious Indexing for Decision-Support in Main Memory. In: Proceedings of VLDB 1999, pp. 475–486. Morgan Kaufmann Publishers Inc. (1999)
20. Rao, J., Ross, K.A.: Making B+-Trees cache conscious in main memory. ACM SIGMOD Rec. **29**(2), 475–486 (2000)
21. Schlegel, B., Gemulla, R., Lehner, W.: k-ary search on modern processors. In: Proceedings of the Fifth International Workshop on Data Management on New Hardware - DaMoN 2009, p. 52. ACM Press, New York (2009)
22. Wulf, W.A., McKee, S.A.: Hitting the memory wall. ACM SIGARCH Comput. Archit. News **23**(1), 20–24 (1995)