High-Performance Tree Indices: Locality matters more than one would think

Thomas Kowalski
Imperial College London
thomas.kowalski19@ic.ac.uk

Fotios Kounelis
Imperial College London
f.kounelis20@ic.ac.uk

Holger Pirk
Imperial College London
hlgr@ic.ac.uk

ABSTRACT

In-memory data management systems have become the state-of-the-art, which leads to the development of highly-efficient index structures. With the elimination of most sources of software overhead, such as interpretation, the impact of low-level hardware parameters becomes more pronounced. Input data locality is such a parameter: It is well known, data locality plays a significant role in indices like B-trees, to our surprise, we found that the impact of data locality on the performance of high-performance indices has not been studied. In this paper, we study the effect of locality on two of the most prominent high-performance index structures: Bw-Trees and Adaptive Radix Trees (ARTs). In our experiments, we find that Bw-trees are highly affected by the locality on inserts (performance difference exceeding a factor two) but fairly robust on lookups. ARTs, in contrast, are moderately affected in their lookup performance (roughly 60%) but robust during insert.

PVLDB Reference Format:
Thomas Kowalski, Fotios Kounelis, Holger Pirk. High-Performance Tree Indices: Locality matters more than one would think. *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

Keywords

Index Structures, Bw-tree, Adaptive Radix Tree, Locality, Performance

1. INTRODUCTION

Indexing is one of the most well-studied topics in data management research [13, 16, 18, 19, 25]. It eliminates the need for linear scans through relations when selecting or joining tuples. Virtually every major database vendor supports some form of indexing, most notably tree- and hash-based approaches. While hash-based structures can locate tuples in constant time (assuming a constant conflict rate), they require rebuilding large fractions of or even the entire hash-table once a tuple is inserted into a hash table that is filled beyond its fill-factor. Tree-based indices, on the other hand, never require more than logarithmically many modifications upon update but suffer from higher (logarithmic) access cost. The robust update behaviour makes them the most popular option for persistent indexing as evidenced by a broad body of research ([4, 23, 29, 1, 22, 10, 12, 27, 7], to name a few). In fact, tree-based index structures are popular in domains beyond classic data-management, such as operating systems [20].

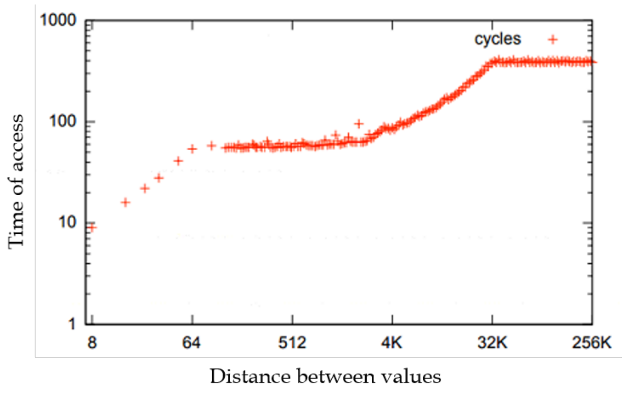

Figure 1: Number of cycles spent based on the locality of the data [17]

Given the current trend towards memory-resident databases, it is not surprising to see many index structures being designed to operate on memory-resident data. Due to the fast storage medium, the key requirements for main-memory indices are high CPU efficiency and support for efficient multi-threaded access. Two of the most prominent main-memory optimized index structures are Adaptive Radix Trees (ARTs) [10] and Bw-Trees [12]. Bw-trees are a latch-free, multi-threaded index derived from classic B-Trees. ARTs, on the other hand, are a trie-based structure, which is primarily designed for CPU efficiency [10]. Which of these two is more appropriate for a workload depends on several parameters: the degree of hardware parallelism, the cost of synchronization instructions, the workload, and many more [5]. One of the parameters that have not yet received significant attention in research is access locality,

11th International Workshop on Accelerating Analytics and Data Management Systems (ADMS'20), August 31, 2020, Tokyo, Japan.
Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

both for inserts as well as lookups.

Locality, i.e., the degree to which access to data items that are close in memory occur close in time[1], has a significant impact on performance. To illustrate the effect, consider Figure 1: it demonstrates that in a simple in-memory array as the distance between accessed values increases, the time per access increases as well. Due to the complex memory hierarchy, it does so in a non-linear fashion. While locality is commonly exploited in high-performance algorithms such as radix-joins [2, 3] or sorting [8] we found that the effect of locality has not been studied in the context of high-performance index structures. To fill that gap, we study the effect of locality for two of the most prominent high-performance index structures and find that locality impacts both, their microarchitectural behaviour as well as the data structure itself.

Specifically, we make the following contributions:

- we conduct extensive experiments studying the impact of parameters such as data distribution, data access locality, bandwidth/compute balance and level of contention (for Bw-trees),
- we find that while both studied index structures are affected by data locality with the specific effect, varying in non-obvious ways,
- to explain the behaviour, we provide and interpret a microarchitectural breakdown for each of the key operations and each of the studied index structures as well as a breakdown into the internal operations where necessary.

The remainder of the paper is structured as follows. In Section 2 we establish the necessary background for the indices we study. In Section 3 we describe the experiments and present their results. In Section 4 we review some recent work focusing on data locality. In Section 5 we conclude and discuss future work.

2. BACKGROUND

Let us, in this section, provide the necessary background knowledge on the two index structures we studied.

2.1 Adaptive Radix Trees

Adaptive Radix Trees (ART) [10] are generalised radix trees (or tries). A radix tree is a compressed trie in which nodes without siblings are merged with their parents. The edges, between these nodes, contain a single element or the sequence of elements. The edges have a label representing the transition *character* to the next node. Nodes contain a flag indicating whether they are terminal i.e. whether a *word* ends at that node. The set of all *words/elements* stored in a radix tree can be computed by reading the sequence of characters on all paths from the root node to each terminal node. In contrast with other tree structures, the radix tree stores data mainly in its edges. The main benefit of such a choice is the lookup complexity, which does not depend on the number of elements in the radix tree but only on the length of the longest *word*. Furthermore, radix trees do not need balancing operations and a naive traversal is always in lexicographical order.

[1]Note that locality is related to but different from data skew

The downside of this structure is the cost of a high memory footprint. The space needed to store a radix tree node is constant and depends on its fan-out. Some of the intermediary nodes do not use all their children's pointers, which results in wasted memory. We consider discussing and addressing the worst-case behaviour, out of the scope of this paper.

ARTs were introduced to solve this memory waste by adapting its nodes to their payload at run time. There are four different types of nodes used by ARTs distinguished by their ability to store more elements or to consume less memory [10]. These four nodes are of size 4, 16, 48, and 256, each holding as many key-child pointer pairs.

There are three different methods to store the payload for a given key. The first is *Single-value leaves*, which adds a new node type that stores exactly one element at the cost of one more pointer. The second is *Multi-value leaves*, that assumes that all keys have the same length and use the same node types for leaves and inner nodes, storing a payload instead of a pointer in the leaves. the last method, *Hybrid nodes*, is only available if values fit in the size of the pointer and use a flag bit to indicate whether a value should be interpreted as a child pointer or as a value.

To further reduce their memory footprint, ARTs implement two kinds of compression: *path compression* and *lazy expansion*. Path compression removes single-child nodes by merging them recursively into their only child, while lazy compression removes paths to "single leaves" by merging them into their first non-single-child parent [10].

2.2 Bw-trees

Bw-trees [12] are a latch-free form of the B-Trees. Their design is motivated by the objective to maximize the degree of parallelism during insert. To do so they use hardware atomic memory operations, such as *compare-and-swap* (*CaS*). Bw-trees also use log-based records instead of in-place storage [15].

The modifications in a node of a Bw-tree are done out-of-place, which means instead of editing the physical node, the modifier prepends a *delta node* to a *delta chain* (a list of events: insertion, deletion, delete, split or merge). Furthermore, nodes are connected using logical links (identifiers rather than pointers). That requires one more indirection but allows for fewer cache invalidations (as the node itself is not modified) and allows latch-free modifications. When a modifying operation prepends a node to the delta chain, it constructs the new node before updating the entry in the mapping table to the new delta node using compare-and-swap. In case of a conflict (two threads trying to update the mapping table at once), one of them aborts and retries. To create a new node, Bw-Trees inherit the splitting behaviour of B-Trees: When a node is full, it is split into two nodes, each containing half of the values.

To get the most recent state of a node, its delta chain needs to be walked. Naturally, walking along the delta chain is costly. To address this, delta chains past a certain length are *consolidated* i.e. applied to the delta node. The physical link is updated and the old node and delta chain is marked for garbage collection. Garbage collection is performed automatically using an epoch-based method, where a resource is freed when all threads have stopped using it.

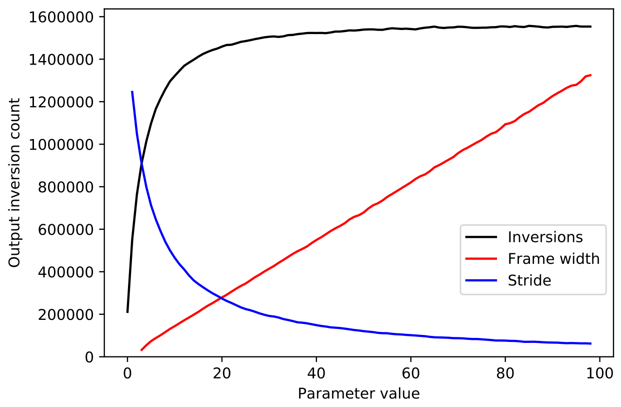

Figure 2: Influence of the algorithm parameters on the total number of inversions created by the locality generation algorithm, starting from a sorted array

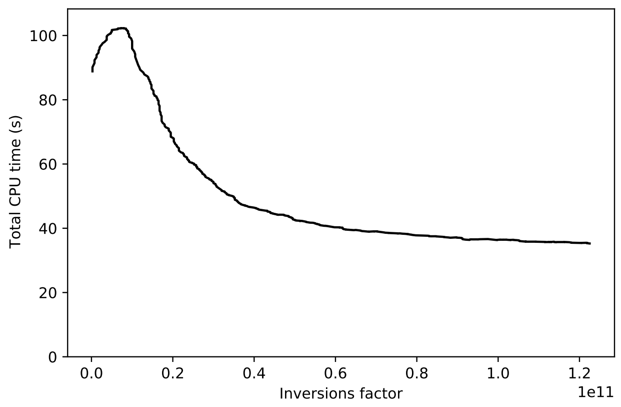

Figure 3: Influence of data locality on the experiment run time for the Bw-tree Insertion workload (8 Threads – 50M Elements)

3. EXPERIMENTAL STUDY

Having discussed the index structures, let us now evaluate the influence of input data locality on their transactional performance [2]: a Bw-tree and both a single-threaded Adaptive Radix Tree, designed for concurrency and thread-safe Adaptive Radix tree. They were chosen as representatives, for state of the art main-memory indices. We study the effect of locality on either by measuring the time taken to process different workloads for two scenarios. The first consists of inserting all elements from the input into the index in the order they appear in the input; the second consists of looking up all items (again, in the order defined by the input) in an already densely populated index. In our experiments, we found that all the results are "symmetrical", which means that reversely sorted inputs have as much locality as the sorted inputs. For clarity, we omit the mirrored section of all charts. Since most experiments sweep a dense parameter space, all result curves have been smoothed using a moving average with window size 15 to mitigate system noise (unless stated otherwise).

3.1 Rationale

Readers may note that Bw-trees and ARTs are fundamentally different data structures, because the first was designed for lock-free concurrency while the latter for CPU efficiency, and can, thus, not be compared directly. There exist a thread-safe version of ARTs [11] which is significantly slower than the classic ARTs, as they have extra overhead for lookup operation, which is not included in the original version. We conduct experiments to both classic and thread-safe ARTs. However, our objective is to evaluate the relative impact of locality. For that reason, we decided to provide the most complete picture by evaluating each of these cases (in the case of the classic ARTs, we run multiple independent instances).

3.2 Setup

The experiments were carried out on an Ubuntu 18.04 LTS instance with an Intel(R) Xeon(R) CPU E5-2660 v3 at 2.60GHz with 32GB RAM. This processor has one thread per core and ten Haswell cores per socket, adding up to a total of 20 threads. We found the influence of locality most pronounced when using 8 threads (most likely due to the hardware being designed towards more CPU-intensive workloads). Profiles and micro-architectural data were obtained using Intel VTune Amplifier.

We used existing implementations of the two indexes:

- For Bw-tree, we used `OpenBwTree` [27]. Note that OpenBwTree is an optimized re-implementation of the original Bw-tree as several design-decisions were not fully documented in the original paper[12].
- For the single-threaded ART, we used `libART` [3].
- For the thread-safe ART, we used `ART Synchronized` [4] [11].

3.3 Data Generation

Studying its impact requires us to carefully control the degree of locality in our input data. This prevents us to use any of the standard datasets but requires that we generate our own using the degree of parallelism as a parameter. Let us briefly describe the process here.

As is common, we use the number of inversions in the input array as a proxy metric for data locality and use an algorithm similar to Fisher-Yates shuffle to (stochastically) control the number of inversions. We initialize an array of n elements with values from a given distribution. For lookup workloads, the distribution is a Zipfian with varying skew factor α. For insertion workloads, using a Zipfian distribution is not possible, as it generates a large number of duplicates, which are not actually inserted in the tree, making comparison impossible. We then slide a window of width f and stride s over the array. At each step, we perform k inversions on randomly selected pairs of elements of the

[2]note that we deliberately exclude bulk-loading optimizations [6] from the scope of this paper

[3]https://github.com/armon/libart

[4]https://github.com/flode/ARTSynchronized

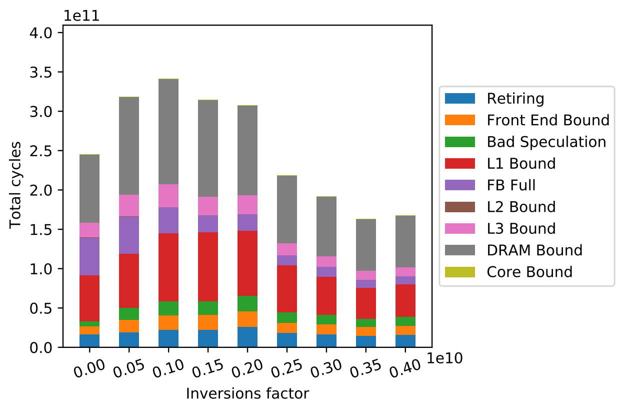

Figure 4: Microarchitectural breakdown for Bw-tree insertion workloads with varying degrees of locality (8 Threads – 50M Elements)

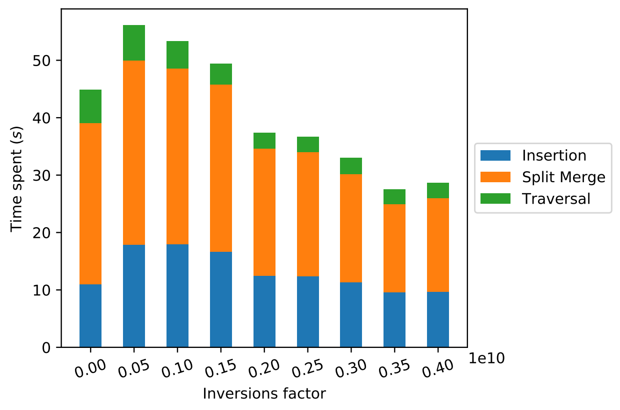

Figure 5: Breakdown of the time spent on each kind of operation when executing the Bw-tree insertion experiment with varying degrees of locality (8 Threads – 50M Elements)

window. Naturally, a larger window produces less locality, with a window of size 1 leading to fully sorted data and a window of size n leading to a fully random array. Figure 2 shows the influence of s, f and and k on the output locality.

However, as the algorithm is not deterministic, the exact output degree of the locality can only be determined by counting the inversions after applying it. Counting inversions in an array, however, is expensive. Fortunately, the number of inversions grows linearly with the window size f (as illustrated by the red line in Figure 2). Consequently, we approximate it from the parameters of the data generation algorithm to speed up experimental runs (we keep the stride $s = 5$ and the number of inversions per stride $k = 1$).

For single-threaded experiments, we create 50 million input in the manner described. For multi-threaded experiments, we create one such input sequence per thread (naturally reducing the elements per sequence by the number of threads).

3.4 Results

Let us, discuss Bw-Trees first, followed by ARTs for either discuss inserts and lookups.

Bw-Trees

Insert. In the first experiment, we study the effect of locality on the insert performance on Bw-Trees. Figure 3 displays the cumulative time to insert all of the uniformly distributed ($\alpha = 0$) input values into the tree. We observe that the execution time increases with the inversion factor between 0.0 and 0.1×10^{11} and decreases after that. This, somewhat unexpected, behaviour leads us to conduct further experiments.

We hypothesize that the increase, in the beginning, is due to an increasing number of cache misses while the decrease for higher inversion factors is due to a reduction in the number of split-merge operations (a behaviour Bw-Trees inherit from B-Trees). To substantiate those hypotheses, we performed two kinds of breakdowns: a microarchitectural breakdown (Figure 4) as well as a breakdown by functional

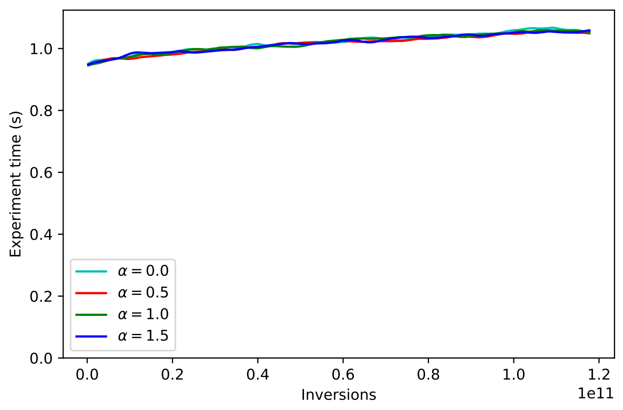

Figure 6: Comparison of run time for Bw-tree Lookup workload with Zipfian distributions of varying α (8 Threads – 50M Elements)

components, i.e., Tree-traversal, Insertion, and Split-Merge (Figure 5). Our analysis is orthogonal to the in-memory OLTP work in [24], as ours is more concentrated on these index structures.

The microarchitectural breakdown (Figure 4) displays, among other things, a high increase of DRAM-related access cost between inversion factor 0.0 and 0.1×10^{11}. This increase in cache-thrashing substantiates our hypothesis that worse cache access behaviour is causing the cost increase. Note that categories in the microarchitectural breakdown do not overlap: they do not represent time spent waiting for each component but rather the proportion of samples in which each component is bounding performance.

Figure 5 shows that the decreasing run time for higher inversion factors can be mostly attributed to a reduction of the split-merge costs. This effect is due to the worst-case splitting behaviour Bw-Trees inherit from B-Trees: since nodes are split when they are full, a fully sorted input (inversion factor 0) leads to values being appended to a single

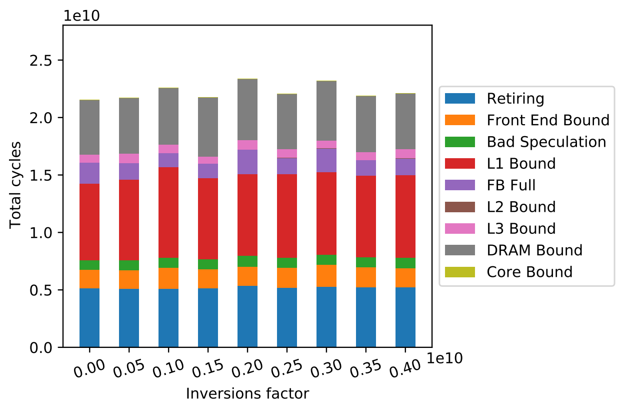

Figure 7: Micro-architectural breakdown for Bw-tree lookup workloads with varying degrees of locality (8 Threads – 50M Elements)

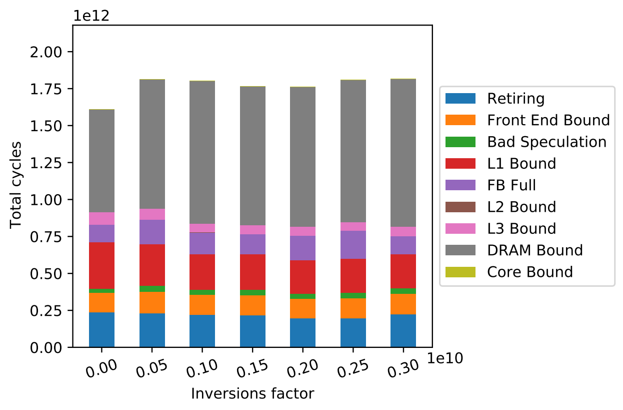

Figure 9: Microarchitectural breakdown for the classic ART Insertion workload with various degrees of locality(8 Threads – 50M Elements)

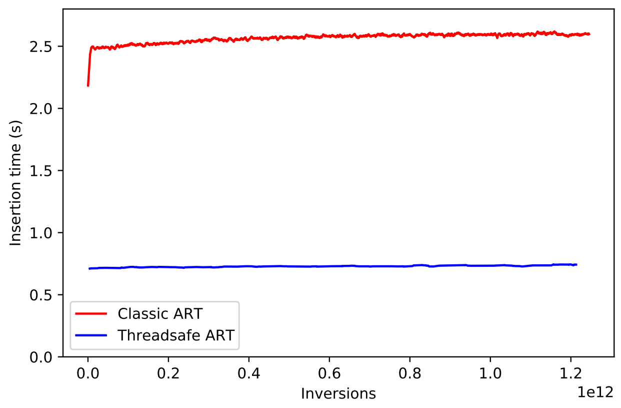

Figure 8: Influence of data locality on the experiment run time for the thread-safe ART and classic ART Insert workload (8 Threads – 50M Elements)

node until it is full at which point the node is split. After the split, all further values are appended to the newly created right-hand node while the left-hand node receives no further values appended and remains only half-full. Consequently, most nodes are only minimally (i.e., half) full in the fully sorted case while the fully random case does not exhibit that (worst-case) behaviour [12]. As a result, data locality affects the structure of the Bw-tree but also affects hardware features, such as cache hits.

Lookup. Figure 6 displays the cumulative time to lookup values from inputs generated with different Zipfian distributions ($\alpha \in \{0.0, 0.5, 1.0, 1.5\}$) in the tree. We observe that the lookup costs grow slightly with the inversion factor. This is in line with our expectation as a larger fraction of the tree is accessed randomly which causes cache thrashing (much like the trend for ART lookups). However, the growth is only slight (roughly 5%) – certainly not the orders of magnitude usually attributed to the cache hierarchy (as in Figure 1).

A microarchitectural breakdown (Figure 7) shows that the profile does not change substantially. In particular, the percentage of DRAM-related costs is relatively small (roughly 15%) and does not increase substantially. Instead, L1-cache-related costs are the most significant cost factor. This is due to the comparatively large node size of the Bw-Tree implementation (128 elements, *i.e.* 512 bytes). The linear traversal of those nodes allows the cache to pre-fetch large fractions of the node which leads to the low impact of access locality. Having covered Bw-Trees, let us, now, investigate ART.

Adaptive Radix Trees

Insert. Figure 8 displays the total time required to insert all input values into a classic and a thread-safe ART. We observe that the execution time, for the classic ART, is virtually unaffected by the inversion factor. This illustrates that ARTs are robust against changes in insert locality. In particular, we expected a more pronounced cost increase for lower inversion factors due to increased cache-thrashing but did not find it in our experiments.

For the thread-safe ARTs, we observe that the inversion factor does not affect the execution time of this operation. As with the classic ARTs, we expected an increase in the cost of sorted values in contrast to fully random values due to cache-thrashing.

To further inspect this behaviour we performed a microarchitectural breakdown of the classic ART experiment and display it in Figure 9. It shows a slight increase in DRAM-relative accesses when increasing the inversion factor from 0.0 to 0.05×10^{10}, but not enough to make a substantial difference in running time.

The thread-safe ARTs are more than x4 faster than the classic ARTs. As a result, locality does not affect the behaviour of the insert operation execution time but it requires less time to run the same experiment than the classic ARTs.

Lookup. In the final experiment, we studied lookup performance for both classic and thread-safe ARTs. Figure 10

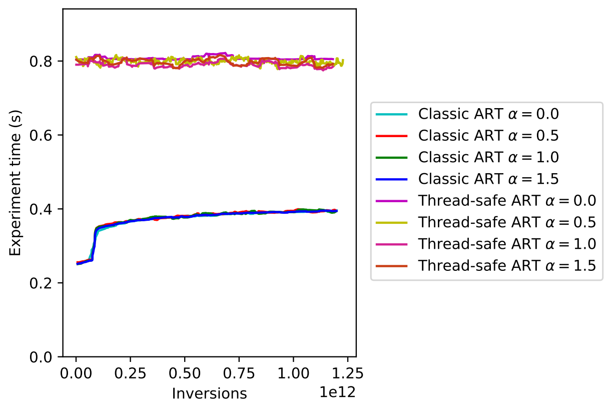

Figure 10: Influence of data locality on the experiment run time for the thread-safe ART and classic ART Lookup workload at different values of α (8 Threads – 50M Elements)

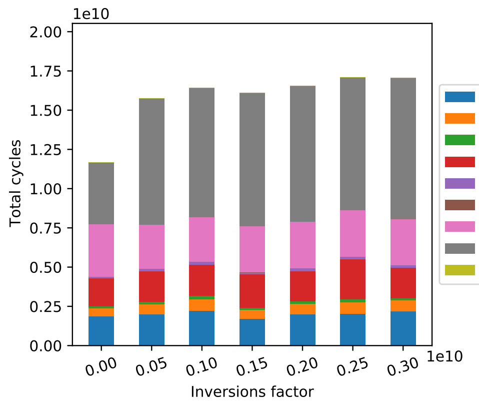

Figure 11: Microarchitectural breakdown for classic ART Lookup workloads with varying degrees of locality (8 Threads – 50M Elements)

displays the cumulative time to look up values from different Zipfian distributions (with $\alpha \in \{0.0, 0.5, 1.0, 1.5\}$) in the tree. First, we observe that execution time is unaffected by skew – apparently skew has little to no impact on performance. Locality, however, has a significant impact: the difference between fully sorted (on the left) and fully random (on the right) is roughly 60%. We also observe that the cost increase is disproportionately high between 0.0 and 0.1×10^{10}.

We observe that the performance of the lookup operation for the thread-safe ARTs is not in line with the classic ARTs in two respects. First, it is robust with respect to locality. Also, the performance of the classic ARTs is better by more than a factor two, than the thread-safe ARTs.

Our hypothesis for that behaviour is that the sharp increase for inversion factor close to 0.0 is due to a higher number of cache access, whereas when the locality decreases, so does the cache hit ratio. To prove our hypothesis, we perform a microarchitectural breakdown: Figure 11 shows that indeed DRAM-related access costs are contributing roughly 25% to the overall runtime for the inversion factor 0.0. The DRAM-related costs more than double when increasing the inversion factor 0.1×10^{10} which substantiates our hypothesis that performance difference is mostly caused by cache locality.

4. RELATED WORK

Before concluding, let us briefly discuss related work on the subject of data locality (we discussed related work on indexing in Section 2). We can broadly distinguish two lines of research: locality-sensitive operators and task scheduling. In the line of locality-sensitive operators, Rödiger et al. [21] study the operators themselves, while Zamanian et al. [28] introduce the idea of predicate-based partitioning to improve data locality. In the line of task scheduling, Muddukrishna et al. [14] use locality for scheduling processes to replace existing scheduling to minimize adversarial NUMA effects. Similarly, Wang et al. [26] propose to organize tasks in queues based on the data size and location. Also, Kumar et al. [9] propose to use spatial locality on cache blocks and introduce a mechanism to predict which portions of the cache block will get used before they get evicted.

5. CONCLUSION AND FUTURE WORK

Data locality has been known as an important parameter for the performance of data processing algorithms such as joins and sort. However, its impact on the construction and use of high-performance index structures has not been sufficiently studied. In this paper, we studied the impact of data locality on two of the most prominent high-performance index structures: Bw-Trees and Adaptive Radix Trees. We found that both are significantly affected by data locality: Bw-Tree insert performance varies by a factor of more than two depending on locality while single-threaded ART lookup performance differs by roughly 60%. These results are the first step towards a better understanding and should extend both in-depth and breadth. For depth, data locality should be studied for a wider range of indices. While to increase the breadth, other more complex kinds of locality such as correlation and auto-correlation on complex data structures such as database indices merit further study.

6. REFERENCES

[1] C. F. Ahmed, S. K. Tanbeer, B. Jeong, and Y. Lee. Efficient Tree Structures for High Utility Pattern Mining in Incremental Databases. *IEEE Transactions on Knowledge and Data Engineering*, 21(12):1708–1721, 2009.

[2] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 362–373. IEEE, 2013.

[3] P. A. Boncz, S. Manegold, M. L. Kersten, et al. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, volume 99, pages 54–65, 1999.

[4] D. Comer. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[5] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. *SIGPLAN Not.*, 47(4):37–48, 2012.

[6] G. Graefe and H. Kuno. Modern b-tree techniques. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1370–1373. IEEE, 2011.

[7] I. Jaluta, S. Sippu, and E. Soisalon-Soininen. Concurrency control and recovery for balanced b-link trees. *The VLDB journal*, 14(2):257–277, 2005.

[8] D. E. Knuth. *The art of computer programming*, volume 3. Pearson Education, 1997.

[9] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No. 98CB36235)*, pages 357–368. IEEE, 1998.

[10] 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.

[11] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The art of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, DaMoN '16, New York, NY, USA, 2016. Association for Computing Machinery.

[12] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 302–313. IEEE, 2013.

[13] Y. Li, B. He, Q. Luo, and K. Yi. Tree indexing on flash disks. In *2009 IEEE 25th International Conference on Data Engineering*, pages 1303–1306. IEEE, 2009.

[14] A. Muddukrishna, P. A. Jonsson, V. Vlassov, and M. Brorsson. Locality-aware task scheduling and data distribution on NUMA systems. In *International Workshop on OpenMP*, pages 156–170. Springer, 2013.

[15] A. Pavlo. Latch-free OLTP Indexes - Part II, 2017.

[16] S. Peng, Y. Yang, Z. Zhang, M. Winslett, and Y. Yu. Dp-tree: indexing multi-dimensional data under differential privacy. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 864–864, 2012.

[17] H. Pirk. Cache Conscious Data Layouting for In-Memory Databases. 2010.

[18] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Prefix hash tree: An indexing data structure over distributed hash tables. In *Proceedings of the 23rd ACM symposium on principles of distributed computing*, volume 37, 2004.

[19] C. Rathgeb and A. Uhl. Iris-biometric hash generation for biometric database indexing. In *2010 20th International Conference on Pattern Recognition*, pages 2848–2851. IEEE, 2010.

[20] O. Rodeh, J. Bacik, and C. Mason. BTRFS: The Linux B-Tree Filesystem. *ACM Trans. Storage*, 9(3), Aug. 2013.

[21] W. Rödiger, T. Mühlbauer, P. Unterbrunner, A. Reiser, A. Kemper, and T. Neumann. Locality-sensitive operators for parallel main-memory database clusters. In *2014 IEEE 30th International Conference on Data Engineering*, pages 592–603. IEEE, 2014.

[22] D. Seo, S. Shin, Y. Kim, H. Jung, and S. K. Song. Dynamic hilbert curve-based B^+-tree to manage frequently updated data in big data applications. *Life Science Journal*, 11(10):454–461, 2014.

[23] A. Shahvarani and H.-A. Jacobsen. A hybrid B^+-tree as solution for in-memory indexing on CPU-GPU heterogeneous computing platforms. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1523–1538, 2016.

[24] U. Sirin, P. Tözün, D. Porobic, and A. Ailamaki. Micro-architectural analysis of in-memory OLTP. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 387–402. ACM, 2016.

[25] J. Wang, W. Liu, S. Kumar, and S.-F. Chang. Learning to hash for indexing big data—a survey. *Proceedings of the IEEE*, 104(1):34–57, 2015.

[26] K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, and I. Raicu. Optimizing load balancing and data-locality with data-aware scheduling. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 119–128. IEEE, 2014.

[27] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen. Building a Bw-Tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, pages 473–488. ACM, 2018.

[28] E. Zamanian, C. Binnig, and A. Salama. Locality-aware partitioning in parallel database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 17–30, 2015.

[29] W. Zhang, H. Tang, S. Byna, and Y. Chen. DART: distributed adaptive radix tree for efficient affix-based keyword search on HPC systems. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–12, 2018.