Contention and Space Management in B-Trees

Adnan Alhomssi
Friedrich-Schiller-Universität Jena
adnan.alhomssi@uni-jena.de

Viktor Leis
Friedrich-Schiller-Universität Jena
viktor.leis@uni-jena.de

ABSTRACT

Classical B-Trees were not designed for the modern hardware. They often do not scale on today's many-core CPUs because they do not handle high-contention workloads well. Moreover, their fixed-size nodes result in underfull nodes and a space utilization below 70%. This low space utilization is economically undesirable, especially with fast but relatively expensive NVMe SSDs, and causes unnecessary read and write amplification. In this paper, we address these two shortcomings by introducing two techniques: The first, Contention Split detects unnecessary contention on nodes and splits them to allow higher concurrency. The second technique, XMerge, saves space by lazily merging neighboring nodes. Both techniques are lightweight, easy to integrate into existing database systems, and result in substantial space and performance improvements.

1 INTRODUCTION

The choice of index data structures in database management systems is greatly influenced by the available hardware. For a long time, database systems were limited by small DRAM sizes and slow magnetic hard disk drives. This led to the classical disk-based system architecture, where data is stored on fixed-size pages and is cached in a DRAM buffer pool. In such a design, the standard index data structure is the B^{+}-Tree because its high fan-out minimizes the number of disk IO requests. When systems with larger DRAM capacities up to a few terabytes became available, many data sets could be kept in main memory, making traditional buffer management unnecessary. This led to the development of pure main-memory database systems like Hekaton [6] and HyPer [17]. In main-memory systems, index data structures are not restricted to fixed-size pages and can use virtual memory pointers instead of buffer pool page identifiers. This new architecture allows new index designs, for example, tries with variable-sized nodes [22].

Now we are witnessing another change in the hardware landscape. Today's many-core commodity CPUs offer as many as 64 cores (128 HyperThreads) on a single socket. PCIe-attached solid-state drives (SSD) have become cheap and fast, while the DRAM prices and capacities have stagnated [12]. Moreover, many data sets are growing beyond the available DRAM capacities. However, extending pure in-memory systems with support for large data sets is difficult. These factors have motivated recent work on high-performance storage engines like Umbra [27] and our LeanStore [20] system. These high-performance storage engines show that buffer managers can be redesigned to allow near in-memory performance when the working set fits into the buffer pool, while supporting large data sets on NVMe SSDs as well.

Because the access latency of SSDs is quite high compared to DRAM (100 µs vs. 100 ns) only balanced data structures with large nodes (i.e., B-trees) are suitable for SSD-optimized storage engines like LeanStore. However, classical B-Trees were not designed for the modern hardware landscape: they scale poorly on today's many-core CPU because they do not handle the physical latch contention that modern CPUs can generate. In previous work [21, 23], we implemented and advocated for *Optimistic Lock Coupling (OLC)* to synchronize B-Trees. OLC eliminates contention generated by readers and allows read-mostly workloads to scale perfectly. However, write contention is still an unsolved problem and a source of performance degradation in real workloads. A second problem with B-trees is space utilization. The average fill factor of B-trees is typically below 70% and can get as low as 39% [14]. This might be acceptable if data is stored on cheap disks but is wasteful on fast NVMe SSDs, which are 5-10 times more expensive than disks.

In this work, we present two lightweight techniques for reducing contention and improving space utilization in B-Trees. *Contention Split* solves the problem of write contention that stems from the fairly large node sizes of B-trees. It first detects unnecessary write contention that is caused by different threads frequently updating different tuples in one node. Then, it splits the node into two to distribute the contention and improve scalability. The second technique, *XMerge*, lazily merges underfull B-tree nodes with their neighbors to improve space utilization. This is done on randomly selected nodes whenever the system runs out of memory. We integrate and evaluate both techniques in LeanStore. Both are also directly applicable to other database systems that use B-Trees. Using TPC-C and micro benchmarks, we show that the two complementary techniques result in substantial performance improvements and space savings.

2 RELATED WORK

As the number of CPU cores keeps increasing, the problem of high contention in OLTP systems is becoming more and more urgent. For this reason, several approaches for mitigating high contention in main-memory systems were proposed [2, 26, 28]. However, these solutions require radical system redesign and are not directly applicable to most existing systems [1]. Latch-free indexes like the KISS-Tree [18] and the Bw-Tree [24] appear promising but also suffer from write contention. The Bw-Tree, for example, prepends delta records to nodes using compare-and-swap instructions. In this latch-free design, threads do not contend on the node's latch but on the head of the delta records list – resulting in similar scalability problems. To the best of our knowledge, Contention Split is the first technique that mitigates unnecessary write contention at the

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2021.
CIDR '21, January 10–13, 2021, Chaminade, USA

B-Tree level and therefore can be incorporated into most existing systems.

B-Trees have many nice properties and there have been many studies on them [8–10]. However, space utilization has always been one of their downsides. Textbook B-Trees guarantee a minimum space utilization of only 50%. LSM-Trees, in contrast, store most of the data in the last layer with 100% space utilization but at the cost of more expensive reads. The B*-Tree as defined by Knuth [19] is a variant of the B-Tree that guarantees 66% instead of 50%. Instead of immediately splitting a node when it gets full, the B*-Tree tries to redistribute the node's keys with its right or left sibling. If both siblings are full then redistribution is not possible, hence it splits the current node and one of its siblings into three 66% full nodes. The B*-Tree's eager key redistribution between sibling and its two-into-three split significantly complicate the implementation and add latency to index operations. We are not aware of any practical system that uses B*-Trees. Compared to the B*-Tree, our proposed XMerge requires fewer CPU cycles per key, is easier to implement, and results in higher space utilization.

3 CONTENTION MANAGEMENT

On today's many-core processors, many software systems fail to scale linearly with the number of cores. Even software that scales well on N cores may scale badly or face performance degradation on 2∗N cores [4]. Despite decades of research, it is difficult to achieve good speedups in fully-fledged systems. Serialized (or non-parallelizable) portions of a program impede scalability. The optimal speedup is inversely proportional to the serialized portion as Amdahl's law states. For example, if we manage to reduce the serialized portion from 2% to 1%, then we can double our theoretical maximum speedup. Therefore, we have to pay attention to avoid any unnecessary serialization in our systems.

In database OLTP workloads, serialization happens because of contention on shared data structures. This is why we will speak of contention instead of serialization from now on. Beside the workload itself, two design choices are the main sources of contention in database systems: latching granularity and the latch design itself. In the following, we address the issue of fixed latching granularity in buffer-managed B-Trees and present our *Contention Split* technique for preventing unnecessary contention. In the Appendix A, we will also discuss a hybrid latch design that combines the scalability of optimistic latches with the robustness of OS mutexes.

3.1 Contention in Buffer-Managed B-Trees

Concurrent data structures in buffer-managed database systems are synchronized at the granularity of a single page. This is in contrast to main-memory systems where the latching granularity can be more flexible to a degree where latching single tuples is possible. Synchronization at page granularity leads to *unnecessary* contention when a certain node has more than one hot tuple that gets updated very frequently by multiple threads. The updating threads will then contend on the same physical latch even if they access different tuples.

On a modern many-core CPU, a single point of heavy contention can impede scaling. For example, TPC-C with 100 warehouses does not scale past 60 threads on our vanilla LeanStore implementation.

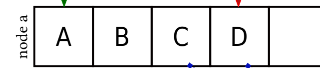

Figure 1: Illustration of Contention Split.

After we profiled the execution run, we found that some nodes have very high contention. The TPC-C *payment* transaction, which constitutes 43% of the workload, updates one of tuples in the *warehouse* relation. Depending on the skew and the number of warehouses, it might happen that a single node stores the tuples of the most used warehouses, or a node has simply enough space for all of them which is the case with 100 warehouses. This serializes all update operations on any *warehouse* tuple and thereby limits the scalability. Note that TPC-C has fairly heavyweight transactions performing dozens of index accesses, the vast majority of which are uncontended. Nevertheless, at today's core counts, Amdahl's law means that even moderate contention may severely degrade overall performance.

There is no straightforward solution to reduce contention in existing systems. Fine-granular latching may reduce contention, but in buffer-managed database systems, the latching granularity is equal to the page size (node size), and a small page size would hurt IO performance. Moreover, the physical organization of pages and tuples is typically not exposed to users. There are no interfaces to detect contention on latches or to manually reorder the tuples in pages. Hence, even expert database administrators have no tools to detect and solve scaling issues systematically.

3.2 Contention Split

Database systems should handle different workload scenarios without manual user tweaks, and high contention is one of them. However, none of the existing systems in literature have an automatic solution to prevent unnecessary contention at the B-Tree level. *Contention Split* is a lightweight mechanism for B-Trees that detects unnecessary contention on nodes. After detection, it splits the contended nodes in order to separate frequently-updated tuples into different nodes protected by different latches. It exploits the fact that, in contrast to tries, the B-Tree structure is quite flexible, i.e., there are many possible B-Trees storing the same data.

Figure 1 illustrates the approach using an example. Thread 1 and 2 frequently update tuples A and D, respectively, and therewith unnecessarily contend on the latch of *node a*. To reduce contention, even though *node a* is not full, we explicitly split *node a* at the middle point between the two updated tuples (A and D), namely at C. Now both threads can update their tuples simultaneously at the cost of an extra node.

Listing 1: Contention Detection and Split

```
struct Page {
  HybridLatch latch;
  // the three probabilistic counters for a period
  int updatesCount; // # updates
  int slowpathCount; // # encountered contention
  int lastUpdatedPos; // last updated tuple position
  unsigned char *pageContent; // B-Tree node
};
// pos: updated tuple position
void afterTupleUpdate(Page& p, int pos, bool isSlowpath) {
  float r = randomNumber(0, 1.0);
  if(r <= sampleProbability) { // track updates
    p.updatesCount++;
    p.slowpathCount += isSlowpath;
    lastUpdatedpos = pos;
  }
  if(r <= periodProbability) { // check for contention
    float ratio = p.slowpathCount/p.updatesCount;
    if(ratio == 1) { // contention detected
      if(pos != lastUpdatePos) { // unnecessary contention
        int splitPos = mid(pos, p.lastUpdatepos);
        btree.splitNodeAtPos(p.pageContent, splitPos);
       }
    }
    p.lastUpdatedPos = -1; // period ends, reset counters
    p.updatesCount = 0; p.slowpathCount = 0;
  }
}
```

The implementation can reuse the existing split operation code that every B-Tree uses for insert operations. The only difference is that we choose a separator halfway between the two frequently-updated tuples, where normally the middle tuple is picked as a separator to make the two new nodes approximately equal in size.

3.3 Contention Detection

One crucial question is how to detect contention without significantly slowing down normal operation. We mark a node as highly contended when most of the worker threads in the current period were not able to exclusively acquire the node's latch immediately, i.e., they had to take the slow path by spinning on the latch or sleeping. A period is not defined by a wall clock time, but rather by a probability. After a thread updates a tuple, we end the current period and start a new one with *period probability*. The reason why we refrain from using time and use a probabilistic approach is elasticity. Finding a good time interval, so we neither run the checks too early nor too late, is very difficult. Our approach, on the other hand, reacts to changes in the workload quickly. If there are many threads updating the same node, it will take a shorter time until a thread hits the period probability and ends the period. If there are few threads, then it will take longer.

During the period, worker threads update three probabilistic counters of the node in which they update a tuple. Probabilistic here means that these counters cover only a sample of the update operations that happened in the last period, and the sample size is determined by *sample probability*. We use sampling to avoid causing additional contention on the counters' cache-line. In Listing 1 we show the C++ code for our algorithm and the probabilistic counters. We found `sampleProbability` = 2^{-2} and `periodProbability` = 2^{-9} to be good settings for small overhead and fast reaction to changes in workload contention. Different settings are evaluated in the Appendix B. When the period ends, the worker thread holding the latch evaluates the counters by calculating how many times the slow path has been taken for every update in the period. If the ratio approaches one, then it treats the node as contended, and compares the tuple position that it has just updated with the `lastUpdatedPos` that another thread has written before. If they are different, then the contention is unnecessary. Hence, it splits the node between the two positions, separating the two frequently-updated keys in two nodes protected by two different latches. As a result, the two tuples can now be updated in parallel.

3.4 Discussion

We have showcased Contention Split only for update operations because we believe that tuple updates are the primary cause for contention in B-Trees in most scenarios. However, Contention Split is also applicable to other tuple-modifying operations like insert and delete. The same logic and pseudocode can be followed after inserting or deleting a tuple in a node.

In this paper, we have focused on latch-based B-Trees. But there are also latch-free B-Tree data structures like the Bw-Tree [24] that promise better scalability but still suffer from contention like their latch-based counterparts [7]. Latch-free B-Trees manage to shorten the critical section of update operations by doing most of the work in thread-local fashion then publishing their update to the shared data structures using atomic compare and swap (CAS) operations. Hence, instead of contending on the node's latch, these latch-free designs contend on another cache-line that holds the atomic word that protects the B-Tree node. When the CAS fails, the updating core has to restart its operation, which resembles optimistic latching. Contention Split can prevent unnecessary contention in latch-free B-Trees just like it does in latch-based ones by splitting nodes and distributing contention on different cache-lines.

With Contention Split, we improve performance at the cost of additional space consumption. At any point in time, the number of nodes with high contention is bounded by the number of cores, bounding the number of splits as well. However, there are cases where contention hot spots move over time. As a result, Contention Split may leave underfull non-contended nodes as time progresses. Moreover, as a consequence of using probabilistic counters, we might, in rare cases, split an uncontended node. To counter these effects and improve the overall space utilization, we propose our *XMerge* technique in the next section.

4 SPACE MANAGEMENT

Textbook B-Trees with fixed-size keys and values guarantee that all nodes, except the root, are at least half full. Space utilization therefore has a lower bound of 50%. The actual utilization depends on the insertion order: for random inserts, the space utilization is about 70% [29]. If the keys are inserted in sorted order, as it is common for primary keys, space utilization remains at the minimum of 50%. In practice, industrial-strength B-Tree implementations diverge from the textbook variant. For instance, they support variable-length keys and extract the common prefix of the keys in a node to accelerate search and save space [25]. Rather than eagerly merging or rebalancing the tree when a node becomes half-empty, they free a node only when it is completely empty [11, 15]. These common

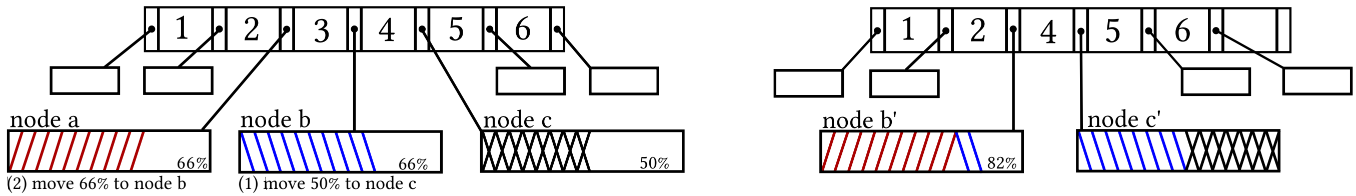

Figure 2: Illustration of using XMerge to merge a group of 3 nodes into 2 to free 1 node.

modifications make space utilizations of 39% or even less possible [14].

In the era of ever growing data sets, space usage on modern secondary storage such as NVMe SSDs is a significant cost factor despite the recent decline in SSD prices. Moreover, low space utilization wastes scarce DRAM resources and can hurt performance. In the following, we propose a novel merge method to improve space utilization in B-Trees, and describe how to integrate it into database systems.

4.1 XMerge

The goal of *XMerge* is to free nodes by merging groups of X immediate siblings into X-1 nodes. When XMerge is called on a node, it searches for a group of X immediate siblings (including the starting node) that have a total free space of at least one node. Once XMerge finds such a group, it compacts all tuples in the group at the right side. This makes the leftmost node empty, so we can reclaim its page. The size of the group X is variable but should not exceed a pre-defined constant K. XMerge tries every group size X from 2 to K and starts compacting the group once the sum of the free space inside is larger than one node. The choice of K forms a trade-off between space utilization and performance. On the one hand, a larger K allows larger groups of nodes that are more likely to have a total free space of one node. On the other hand, compacting more nodes takes time and reduces the concurrency because the parent node gets exclusively latched during the process. We found K=5 to be a good setting for a wide variety of data sets and workloads. In the Appendix B, we evalute different K values for XMerge.

An example of XMerge with K=5 and X=3 is illustrated in Figure 2. We start at *node a* on the left and add the next right node to the group, so the sum of their free space becomes 34 + 34 = 68% which is less than a node. Hence, we continue and add *node c*. Now, the sum is 118% of a node, so we stop looking into other nodes and start compacting: first, we move tuples from *node b* to *node c* as much as *c* fits. This leaves *node b* around 84% empty which is enough to store all tuples from *node a*. Thus, we fully merge *node a* into *node b* and remove *a*'s separator from the parent node. The result is one free node, one full node, and one 82% filled node.

Listing 2 shows the pseudo-code for XMerge, which consists of two phases. In the first phase, we determine the size of the group X. In the second phase, we compact the tuples at the right side then free the starting node.

Listing 2: XMerge

```
const int K = 5; // maximum number of nodes to merge
void XMerge(BTreeNode childNodes[], int startPos) {
  int X = 1; // size of the group
  int totalFreeSpace = childNodes[startPos].freeSpace;
  while(X < K && totalFreeSpace < NODE_SIZE) {
    freeSpace += childNodes[startPos + X].freeSpace;
    X++; // expand the group by one node to the right
  }
  if(totalFreeSpace >= NODE_SIZE) { // can free a node
    int right = startPos + X - 1;
    while(right > startPos) {
      int left = right - 1;
      moveTuples(childNodes[left], childNodes[right]);
      right--;
    }
    freeNode(nodes[startPos]);
  }
}
```

4.2 Where to XMerge?

To integrate XMerge into a database system, we first have to decide which nodes to run XMerge on. It is often the case that index accesses on keys exhibit a pattern: some keys are only read while some are also updated or inserted. As a result, we can see the B-Tree key space as a concatenation of read-only and write key ranges. Ideally, to save space without degrading performance, we would want to compact read-only key ranges next to each other while keeping write key ranges intact. In practice, we do not know in advance where these read-only ranges are. However, in many B-Trees, the read-only key ranges represent the bulk of the B-Tree. Therefore, we run XMerge on *randomly* selected nodes to improve the space utilization in all nodes. The probability that XMerge hits a hot node, in which worker threads are inserting new keys, is low because we pick nodes randomly. If XMerge happens to put frequently-updated key ranges together in one node, then Contention Split will quickly undo the merge and fix any resulting performance degradation.

4.3 When to XMerge?

One naive way to integrate XMerge is to run XMerge in an infinite loop in a background thread. The problem with this approach is that it wastes CPU cycles when the space utilization is already high. Therefore, we have to carefully decide when to run XMerge.

In buffer-managed systems, we want to run XMerge under memory pressure, i.e., when the buffer pool has no more free space. The responsible component for freeing buffer space when needed is the replacement strategy. From the replacement strategy's perspective, it does not matter whether the free space comes from evicting pages or from freeing buffered nodes using XMerge. Therefore, it makes sense to attach XMerge to the replacement strategy. Before a worker thread evicts a page, it picks a random page from the buffer pool and runs XMerge on it with one slight modification. XMerge merges only buffered siblings and never loads an evicted page. If

XMerge manages to free one page, then the thread simply uses the freed page by XMerge. Otherwise, it evicts the next victim page that the replacements strategy points to.

Unlike classical B-Tree merge and rebalance methods, XMerge has no impact on the latency of fast in-memory index accesses. XMerge is only called during index accesses that have to evict a page before reading from disk, which is already a slow process. Moreover, as we show in the evaluation, XMerge has little overhead on throughput.

4.4 XMerge in LeanStore

We integrated XMerge into LeanStore [20]. Our implementation is slightly different than previously described because LeanStore diverges from the traditional architecture and uses a different type of replacement strategy. The buffer pool is divided into hot, cooling and free areas, and the system tries to maintain a fixed percentage for each area using background threads [13]. When the buffer pool runs out of free memory, the background threads evict the oldest pages from the cooling area and move the same amount of pages from the hot to the cooling area. Worker threads focus only on transaction processing and allocate buffer space from the free area without consulting the replacement strategy. Because the pages that the background threads move from the hot to the cooling area are selected randomly, we run XMerge on these pages and move them to the cooling area only when XMerge does not manage to free a page. To summarize, LeanStore runs XMerge in background threads on randomly selected nodes. Other systems can integrate XMerge differently depending on their replacement strategy.

5 EVALUATION

We experimentally evaluate the performance and space savings gained by integrating Contention Split and XMerge techniques into our storage engine LeanStore. Our baseline B$^+$-Tree supports variable-length keys and prefix compression and merges underfull nodes with neighbors on deletion. LeanStore does not yet implement transaction isolation but does synchronize its B-Trees using Optimistic Lock Coupling [21, 23]. We start by showing the impact of each technique separately using TPC-C and micro benchmarks. Then, we show how our two techniques complement each other in an end-to-end evaluation using TPC-C and a synthetic benchmark.

We run all benchmarks on a single-socket system with AMD EPYC 7702P 64-Core Processor (128 hardware threads) running at 2.0 GHz with 256 GiB of RAM. For persistence, we use four 1.6 TB Huawei ES3600P V5 NVMe SSDs with specified 3,500 MB/s read and 1,900 MB/s write throughput per SSD. The SSDs are logically combined using software RAID-0. We used Ubuntu 19.10 as operating system, stored our binaries on a separate SSD, and set the page size to 16 KiB. The source code is available at leanstore.io.

5.1 Contention Split

In the first experiment, we run TPC-C using 100 warehouses, a large buffer pool to encompass all of it, and a variable number of threads. The results are shown in Figure 3. Without Contention Split, the workload stops scaling at 60 threads because of the high contention on nodes that store hot frequently-updated tuples. Contention Split successfully identified and split these nodes. These splits distribute

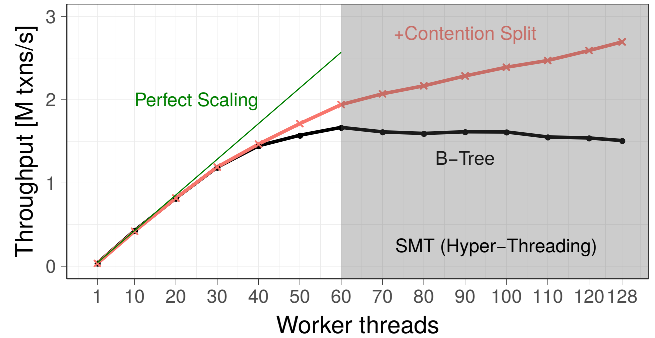

Figure 3: In-memory TPC-C with 100 warehouses.

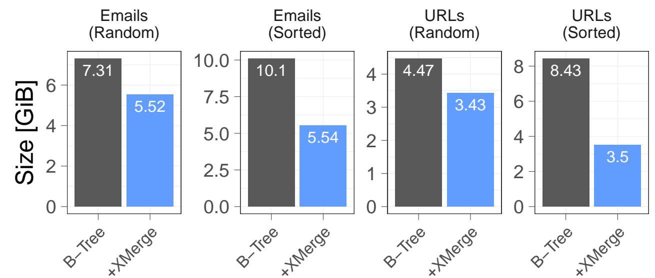

Figure 4: B-Tree size for real-world data.

the contention across different latches which allow higher concurrency and linear scaling. We repeated the experiment on two different machines provisioned from AWS EC2: a dual-socket Intel Xeon 8275CL with 48 cores (96 HyperThreads) and an ARM-based AWS Graviton2 with 64 cores (no HyperThreading). The resulting curves are similar to Figure 3.

We also measured the performance and space overhead of Contention Split on low-contention workloads. We found that the impact on space is negligible and the performance penalty is at most 1.1%.

5.2 XMerge

We first evaluate XMerge using TPC-C and compare the storage cost of each warehouse (i.e., initial data) and new order (i.e., newly inserted data):

		B-Tree	+XMerge
Initial data	MiB per Warehouse	182.2	114.9
New data	KiB per New Order	2.16	0.795

XMerge reduces the storage cost of the initial data by 37% and each *new order* transaction by 63%. We would like to note that simpler approaches that recognize sorted inserts and create new nodes instead of splitting at half may not work here because TPC-C has many sorted insertion points, e.g., one for each combination of *warehouse* and *district* in the *order* relation. We believe that multiple insertion points are common in real-world workloads.

The next experiment compares the storage and CPU costs of inserting a 20 GiB synthetic data set. Keys are 8-byte integers, values are 100 bytes, and the insertion order is random:

	B-Tree	B*-Tree	B-Tree + XMerge
Total size in GiB	31.3	26.6	25.1
Instructions per key	2870	3594	2995
Cycles per key	3497	4580	3645

The B*-Tree produces a compact tree but requires many more instructions than the B-Tree because of the frequent rebalancing between neighboring nodes. XMerge achieves the smallest size among competitors while requiring only around 125 instructions per key in addition to the B-Tree insert. On average, XMerge costs 4 cycles to free one byte, which is in the same order of magnitude as the cost of writing one byte to SSD. The exact cost of writing to SSD depends on the used IO stack. For example, IO stacks with OS buffering and synchronous IO cost about 7 cycles per byte [12], while stacks that use `O_DIRECT` cost one cycle.

We also measure the storage size using two real-world data sets: URLs (3.8 GiB in text) and Emails (3.0 GiB in text). The value is simply a random 8-byte integer, and the insertion order is random. Figure 4 shows that XMerge produces a compact B-Tree regardless of the insertion order. The baseline B-Tree exhibits low space utilization when keys are inserted in sorted order because of the following: The insertion point of sorted keys always moves to the right. The rightmost node has no upper bound and hence cannot use prefix compression, and when it becomes full, the B-Tree splits it at the middle. This leaves less than half-full nodes behind because the new node left of the insertion point can now use prefix compression. When XMerge encounters these underfull nodes, it realizes that prefix compression would make them fit in one node and merges them back. The end result is a very space efficient B-Tree, which is even smaller than the original text in the case of "URLs (sorted)".

5.3 End-to-End Evaluation

All experiments so far focused on one of the two techniques individually. Next, we show how XMerge and Contention Split complement each other using TPC-C and a synthetic benchmark. For our TPC-C evaluation, we fix the size of the buffer pool to 240 GiB, use 120 worker threads, and vary the number of warehouses from 10 to 10,000 which spans in-memory and out-of-memory scenarios. We split Figure 5 into two regions: left of the vertical line where the working set fits in main-memory, and right of the line where it does not. In the in-memory case, Contention Split eliminates write contention points when large number of cores work on small number of warehouses and allows us to fully utilize our CPU. In the out-of-memory case, XMerge relieves the IO subsystem by reducing read and write amplification and shrinks the TPC-C working set size. This allows a larger portion of the working set to fit into main-memory and improves the performance even in the extreme out-of-memory case with 10K warehouses (1.8 TiB) by 1.18× from 55.6K to 66K txns/s (hard to see in the figure).

In the last experiment, we measure the performance using a synthetic benchmark that models a news website (a session store would be similar). We store articles with view counts in a B-Tree index where the key is a sequentially generated 8-byte integer and

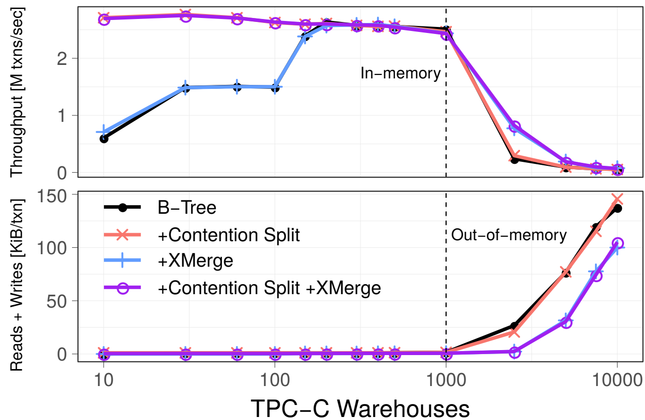

Figure 5: TPC-C performance and IO per transaction using different data sizes, 240 GiB buffer pool and 120 threads.

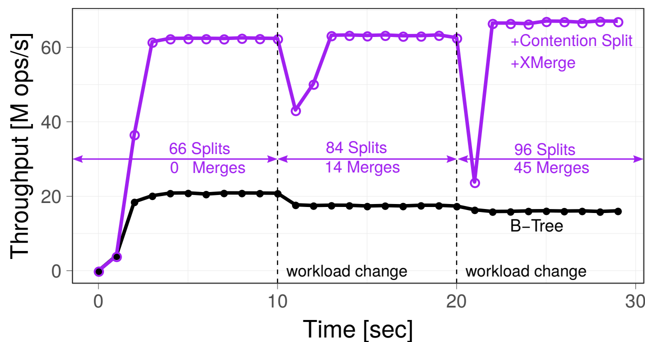

Figure 6: Performance of 50% updates/lookups on recently-inserted tuples. New tuples are inserted every 10 seconds.

the value is 128 bytes. We use 120 worker threads, half of which read an article and half of which update the view count of an article. Only the last 1 GiB of tuples are accessed using the *Latest* distribution from YCSB [5] and a Zipfian factor of 0.75. The more recent an article is, the more likely it is accessed. The buffer pool is big enough to hold the working set in memory. Every ten seconds, we insert 0.1 GiB of tuples which become the new hottest tuples. The results are shown in Figure 6. Our method detects contention timely (in less than 3s), splits the pages and therewith more than doubles the overall average performance. Furthermore, as the contention points move away, the old split and now cold pages are merged back by XMerge—avoiding wasted memory.

6 SUMMARY

We presented two techniques for B-Trees: Contention Split detects unnecessary contention on nodes and splits them to allow higher concurrency. XMerge saves space by lazily merging neighboring nodes together. While the former improves performance and the latter mainly saves space, the two techniques also complement each other: Nodes that are not contended will be eventually merged, and contended nodes will be split. Moreover, both techniques are easy to integrate into existing database systems and result in substantial space and performance improvements.

REFERENCES

[1] Raja Appuswamy, Angelos-Christos G. Anadiotis, Danica Porobic, Mustafa Iman, and Anastasia Ailamaki. 2017. Analyzing the Impact of System Architecture on the Scalability of OLTP Engines for High-Contention Workloads. *VLDB* (2017).

[2] Tiemo Bang, Ismail Oukid, Norman May, Ilia Petrov, and Carsten Binnig. 2020. Robust Performance of Main Memory Data Structures by Configuration. In *SIGMOD*.

[3] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2020. Scalable and robust latches for database systems. In *DaMoN*.

[4] Silas Boyd-Wickizer, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2012. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*.

[5] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*.

[6] Cristian Diaconu et al. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*.

[7] Jose M. Faleiro and Daniel J. Abadi. 2017. Latch-free Synchronization in Database Systems: Silver Bullet or Fool's Gold?. In *CIDR*.

[8] Nikolaus Glombiewski, Bernhard Seeger, and Goetz Graefe. 2019. Waves of Misery After Index Creation. In *Datenbanksysteme für Business, Technologie und Web (BTW 2019)*.

[9] Goetz Graefe. 2003. Sorting And Indexing With Partitioned B-Trees. In *CIDR 2003*.

[10] Goetz Graefe. 2011. Modern B-Tree Techniques. *Found. Trends Databases* (2011).

[11] Jim Gray and Andreas Reuter. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.

[12] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR*.

[13] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *SIGMOD*.

[14] Theodore Johnson and Dennis E. Shasha. 1989. Utilization of B-trees with Inserts, Deletes and Modifies. In *Symposium on Principles of Database Systems*.

[15] Theodore Johnson and Dennis E. Shasha. 1993. B-Trees with Inserts and Deletes: Why Free-at-Empty Is Better Than Merge-at-Half. *J. Comput. Syst. Sci.* (1993).

[16] Anre Kashyap. 2020. *High Performance Computing: Tuning Guide for AMD EPYC 7002 Series Processors*. Technical Report. AMD. https://developer.amd.com/wp-content/resources/56827-1-0.pdf

[17] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*.

[18] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. 2012. KISS-Tree: smart latch-free in-memory indexing on modern architectures. In *DaMoN*.

[19] Donald E Knuth. 1973. The Art of Computer Programming, Volume 3: Searching and Sorting. *Addison-Westley Publishing Company: Reading, MA* (1973).

[20] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-memory data management beyond main memory. In *ICDE*.

[21] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* (2019).

[22] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*.

[23] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *DaMoN*.

[24] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*.

[25] David B. Lomet. 2001. The Evolution of Effective B-tree: Page Organization and Techniques: A Personal Account. *SIGMOD Rec.* (2001).

[26] Ajit Mathew and Changwoo Min. 2020. HydraList: A Scalable In-Memory Index Using Asynchronous Updates and Partial Replication. *VLDB* 13, 9 (2020).

[27] Thomas Neumann and Michael Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*.

[28] Guna Prasaad, Alvin Cheung, and Dan Suciu. 2020. Handling Highly Contended OLTP Workloads Using Fast Dynamic Partitioning. In *SIGMOD*.

[29] Andrew Chi-Chih Yao. 1978. On Random 2-3 Trees. *Acta Informatica* 9 (1978).

A ROBUST HYBRID LATCHES

The impact of contention on performance depends on the implementation of the latches used to synchronize the database data structures. In LeanStore, we adopted a hybrid latch design that combines the scalability of optimistic latches with the robustness of OS mutexes. Similar to the HybridLock [3], we use an OS read-/write mutex that protects an atomic version integer. This gives us effectively three synchronization modes: optimistic for point reads, pessimistic shared for scans, and pessimistic exclusive for writes. An atomic version is used to synchronize with optimistic readers and the mutex provides the pessimistic exclusive and shared modes. After acquiring and before releasing the mutex in exclusive mode, the holding thread increments the atomic version integer by one to notify optimistic readers with changes.

The main advantage of adopting such a hybrid design is the kernel-supported contention handling strategy of the OS mutex that puts contending threads to sleep in a queue as long as the latch is acquired by another thread. We refer the reader to the paper of Böttcher et al. [3] for a detailed description and evaluation of the hybrid latch and other busy-waiting latches. In this section, we emphasize on the importance of implementing robust latches and avoiding busy-waiting in scalable databases – particular given the fact that even single-socket systems are becoming cache-coherent NUMA (ccNUMA). Although spinlocks can achieve better peak performance under low contention, they can lead to performance degradation [3, 4] and unfairness in database systems as we show in the next experiments.

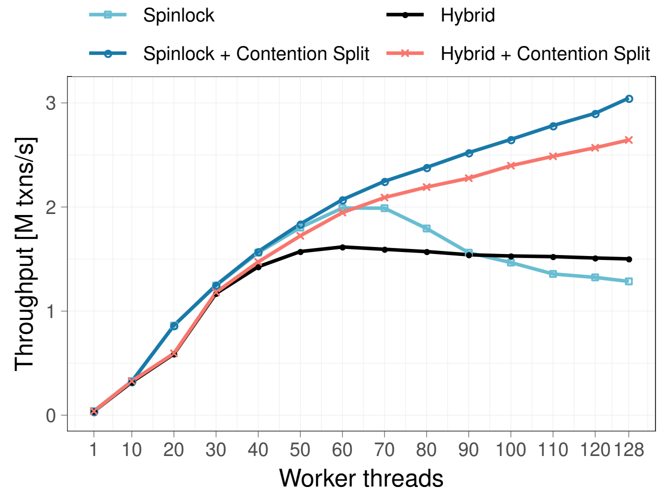

Figure 7: In-memory TPC-C Performance with 100 warehouses using different latch implementations.

A.1 Performance Degradation

We run in-memory TPC-C with 100 warehouses and two different latch implementations: our *Hybrid* latch and a *Spinlock* with optimistic mode and exponential backoff strategy. We also enable Contention Split for both latches and show the results in Figure 7. Although the Spinlock variant achieves the best peak performance when Contention Split takes care of unnecessary contention, its performance degrade in the vanilla B-Tree without Contention Split as more threads contended on the Spinlocks. The performance degradation happens because the critical section becomes longer and requires more serial cycles as more cores spin on the cache-line of the Spinlock [4]. The same applies for other busy-waiting lock variants. The hybrid latch without Contention Split does not scale beyond 50 but at least it puts the contending threads to sleep instead of polluting the CPU caches and degrading the performance

which is essential property for robust systems. The combination of hybrid latch and Contention Split gives us the best balance between robustness and performance.

A.2 Unfairness

Modern many-core CPU are becoming ccNUMA with multiple dies in one package. This means that even single-socket systems are not Symmetric Multiprocessing (SMP) systems anymore and the internal physical organization of cores will be reflected in NUMA domains. We take our EPYC 7702P CPU [16] as an example and show how its multi-chip design affects the fairness of busy-waiting latches.

Our CPU consists of 4 quadrants, each quadrant is a group of 2 Core Complex Die (CCD) and each CCD contains two Core Complex (CCX) which in turn contains 4 Zen2 cores and a separate L3 cache. Memory accesses between cores of different CCDs have higher latency because they communicate over an "Infinity Fabric" interconnect. We configure the processor as single NUMA domain in BIOS and let all the 64 cores (no SMT) of our CPU repeatedly update the same tuple in a B-Tree and measure the percentage of updates that each core manages.

The results in Table 1 are grouped by CCD. The hybrid latch is fair among cores whereas the Spinlock with and without backoff is not. We observed that busy-waiting schemes make it more likely that the next core to update the tuple lies in the same CCX or in the same quadrant of the core that has just updated the tuple.

Table 1: Percentage of updates that each Core Complex Die (CCD) of an EPYC 7702P manages using different latches.

CCD	1	2	3	4	5	6	7	8
Spinlock	0.8	0.9	46.2	50.9	0.2	0.2	0.5	0.4
+Backoff	13.6	13.4	21.4	21.6	7.1	7.1	8.0	8.0
Hybrid	12.7	12.5	12.7	12.5	12.5	12.3	12.3	12.4

To summarize, we avoid read contention using optimistic latches, eliminate unnecessary write contention using Contention Split, and guarantee robustness and fairness when contention is inevitable using the pessimistic exclusive and shared modes of OS read/write mutexes.

B PARAMETERS

The two techniques we introduced in this paper require few parameters. In this section, we empirically evaluate different settings and show their impact on performance and space usage.

We start with Contention Split which takes two parameters: `sampleProbability` and `periodProbability`. The first parameter determines the percentage of the update operations that are covered in the sample probabilistic counters. Frequently updating the counters gives us more representative sample but cost all update operations, including the non-contended ones, more CPU cycles. We evaluated different values for `sampleProbability` and settled on 25.0% = 2^{-2} because it gives a good quality sample of the recent updates while causing negligible overhead to non-contended workloads.

The second parameter controls the period length. In Contention Split, we want to detect highly and unnecessarily contended nodes.

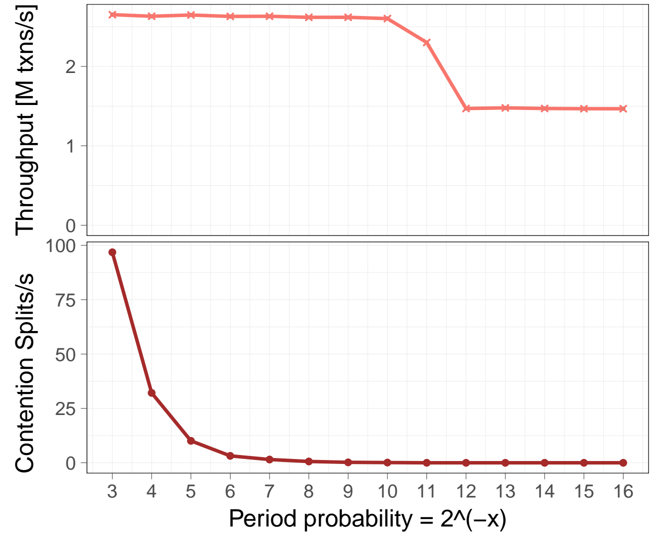

Figure 8: Contention splits rate and TPC-C performance using different period probabilities.

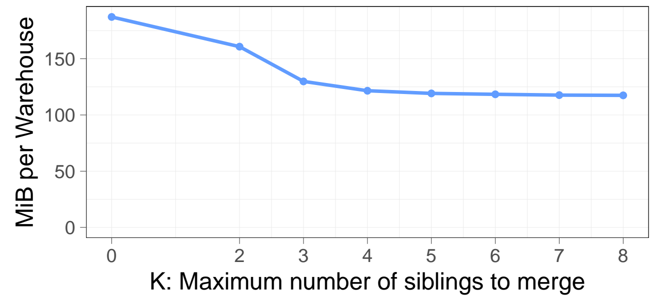

Figure 9: Size of a TPC-C warehouse using different K constants for XMerge.

Thus, the period must be long enough to make sure that contention on the node is really high and short enough to timely react to changes in the workload. To illustrate the impact of this parameter on performance and the number of split nodes, we run in-memory TPC-C with 100 warehouses and 128 threads using fixed sample probability and different period probabilities that should be smaller than the sample probability. The results for values between 2^{-3} and 2^{-16} are shown in Figure 8. Large periods, at the beginning of the x-axis, lead to lots of node splits that are false positives. Short periods, at the end of the x-axis, do not detect the contention. Values in the middle, like 2^{-9} which is our default, improve performance and avoid unnecessary splits.

Our XMerge technique takes only one parameter K that determines the maximum number of immediate siblings that we can merge. In all our of experiments, we set K to 5 which is, for instance, enough to empty and free one node in a B-Tree with randomly inserted keys and presumed average fill factor of 70% [29]. Figure 9 shows the impact of the constant K on the storage cost of a TPC-C warehouse initial data, showing that larger values offer little benefit in terms of space reductions.