Contention and Space Management in B-Trees

Seminar: Implementation Techniques for Main Memory Database Systems (Winter Term 2021/2022)

CHRISTOPH ROTTE

The B-Tree and especially its variant, the B$^+$-Tree, are common data structures for indexes used in database systems. Their application in practice, however, may entail performance problems, for example scaling difficulties with the amount of data or number of worker threads.
In this seminar paper, we look into the work of Adnan Alhomssi and Viktor Leis [1] who introduced two techniques, "Contention Split" and "X-Merge", to counteract such problems, more precisely, contention on nodes in multithreaded environments as well as page evictions when using page buffers. While Contention Split lowers the number of simultaneous exclusive accesses on nodes by splitting them, X-Merge works against page evictions by reusing unoccupied space in the B-Tree. Our evaluation shows that both approaches, especially in combination, can increase the overall throughput of a simple database system.

1 B-TREES IN DATABASES

A B-Tree is a balanced search tree with fixed-sized nodes. Choosing a larger constant size helps to limit the height of the tree while maintaining logarithmic operation times. This makes the B-Tree a common choice for indexes in database systems, although, when used in practice, its variant, the B$^+$-Tree is often favored over the textbook B-Tree. In contrast to a B-Tree, the B$^+$-Tree stores the data pointers behind the keys only in the leaf nodes, thus fanning out the structure.
One advantage of the B$^+$-Tree over the B-Tree shows when using systems where the data does not fully fit into the main memory. In this case, we split the memory into equally large pages. A page buffer is then responsible for keeping the currently needed pages in memory by applying a replacement strategy and swapping pages in and out of the disk. We can implement B-Trees and B$^+$-Trees in such a buffered system in the following way:
Since both the buffer pages and the tree nodes need to have a constant size, we map exactly one node onto one page. This allows for an easy organization of the data structure in the buffer. Because the inner nodes of a B$^+$-Tree are denser and contain more node pointers than those of a B-Tree, usually fewer page requests and therefore costly IO-operations occur.
We can achieve synchronized access to the tree in environments with multiple threads by protecting the pages with locks. One trade-off between granular access control and avoiding complex synchronization mechanisms like lock-free approaches such as the KISS-Tree [3] is to keep each page with one lock having both a shared and an exclusive access mode. Threads that want to perform an operation and interact with the data structure thus have to request synchronized access to the locks in addition to the actual node pages from the buffer.
Figure 1 shows an example of such a B$^+$-Tree with page nodes synchronized by locks. Since the focus of this paper primarily lies on a simple database system, we do not take techniques like Optimistic Lock Coupling [5] into consideration which optimize lock-based approaches further.

2 CONTENTION SPLIT

In certain situations, lock-based synchronization approaches come with performance problems. One example of this is contention caused by different threads simultaneously trying to access the same node exclusively. Figure 2 illustrates how node contention can impact the performance of the system. Here, Thread 1 and Thread 2 try to exclusively access data behind two different keys on the same node. Since there is only one lock determining which thread gains access to the node, eventually, one thread will have to wait for the other one to finish, either by spinning or dequeuing. In both cases, we cannot achieve the scaled performance theoretically gained by using multiple threads.
"Contention Split", introduced by Alhomssi and Leis [1], is one technique to counteract such situations in which

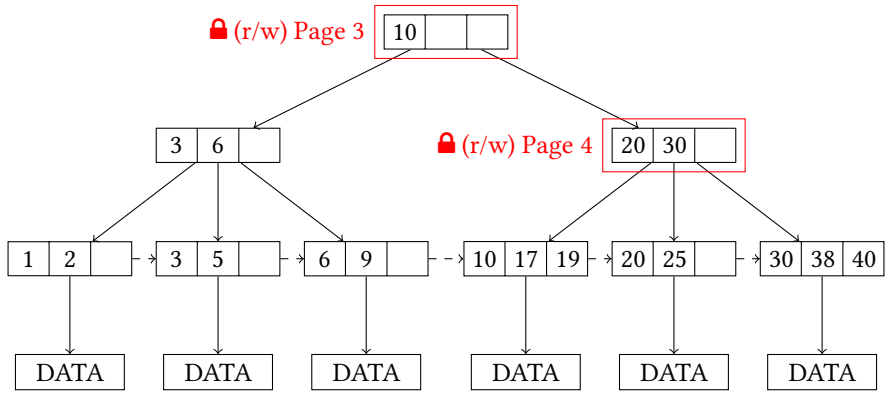

Fig. 1. Exemplary B$^+$-Tree mapped to Buffered Pages with Access Locks

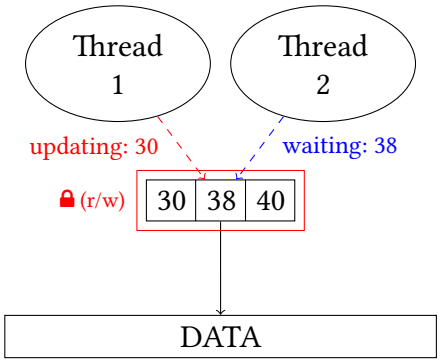

Fig. 2. Illustration of Node Contention

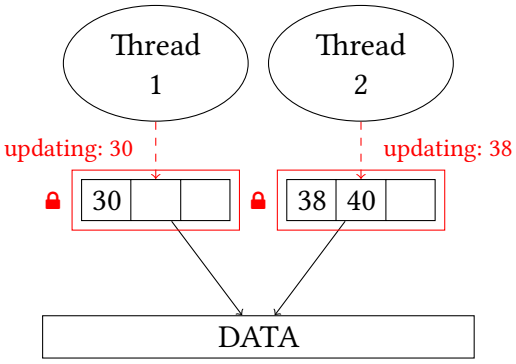

Fig. 3. Simultaneous Access after Contention Split

threads wait on each other to finish. The idea here is to split nodes that are affected by contention if the accessed key indexes are different.

Figure 4 shows the algorithm implemented for the update operation of a B-Tree. After a successful update, the respective thread calls the procedure and generates a random number to test against two constants called *sample_prob* and *period_prob*. The two thresholds work as a heuristic to detect contention happening on the current node page. If our number is below *sample_prob*, we update the number of operations that occurred on the node, the most recently accessed key as well as the number of times a thread had to wait. These values are stored alongside the page. Additionally, if our number is also below *period_prob*, we check the current situation for a contention. If the number of times a thread has had to wait is approaching the number of total operations, we assume that we can eliminate future update delays by splitting the node between the current key and the key that was previously accessed.

Figure 3 illustrates the result of Contention Split after the situation in Figure 2. One of the two threads successfully detected the ongoing contention and split the node between keys 30 and 38. Now, it is possible to gain exclusive access to each key simultaneously.

```
1:  procedure POST_UPDATE (page, update_index, waited)
2:      last_update ← page.last_index
3:      r ← random(0.0, 1.0)
4:      if r < sample_prob then
5:          Update update_count, last_index, wait_count on page
6:      end if
7:      if r < period_prob then  # period_prob < sample_prob
8:          if page.wait_times ≈ page.update_times then
9:              Split page.node at mid(update_index, last_update)
10:             Reset update_count, last_index, wait_count on page
11:         end if
12:     end if
13: end procedure
```

Fig. 4. Contention Split Procedure

3 X-MERGE

As outlined above, page buffers need to employ a replacement strategy to minimize the number of page evictions and therefore IO-operations that generally represent one of the main bottlenecks for database systems.
In their paper [1], Alhomssi and Leis propose a second technique to further optimize preexisting replacement strategies, called "X-Merge". With X-Merge the page buffer directly interacts with the data structure in memory. Since B-Tree nodes have a constant number of key slots that are not fully occupied most of the time, we can merge adjacent nodes and use the freed space to load in a requested page.
Before evicting a page to swap in a requested page, we call the *pre_eviction* procedure (shown in Figure 5). The page buffer then selects a random inner node of the B-Tree and checks whether it is eligible for X-Merge, that is, not currently accessed or pinned by any thread. After that, the buffer loops over a maximum of *max_nodes* children of the node, starting at a randomly chosen index. If the sum of unoccupied slots is enough for the requested page, the selected nodes are merged from left to right. The space of the first node, which was subsequently freed, can then be used to load the requested page.
Figure 6 illustrates the exemplary part of a B-Tree which a page buffer could use for X-Merge. After randomly selecting the top node, X-Merge detects that its three node children have four unoccupied slots, which, in combination with their parent, is enough to free one node. The first child can thus be merged into its two neighbors (shown in Figure 7) and the remaining space used to load the page, thus preventing an IO-write.

4 IMPLEMENTATION AND EVALUATION

To test the performance impact of Contention Split and X-Merge, we ran two different Yahoo Cloud Serving Benchmark (YCSB) [2] workloads on a simple database system that uses a B$^+$-Tree as its primary index. More specifically, we use YCSB-cpp [6], an implementation of YCSB in C++, to easily integrate our system structure into the evaluation environment of YCSB.
The system uses a buffer that employs the clock algorithm [7] as a second-chance replacement strategy with X-Merge and uses fixed-sized pages. Its B$^+$-Tree consists of node pages with single locks and applies Contention Split on respective nodes after each successful update operation. The 1000B tuples behind the keys are stored in frames which we organize in a single file using a freelist. All tests were performed on Manjaro Linux x86_64 with an AMD Ryzen 5 2600X Six-Core Processor (3.6GHz, 12 hardware threads) and a Samsung 860 EVO SSD (sequential reads: ~550 MB/s, sequential writes: ~520 MB/s).

```
1:  procedure PRE_EVICTION (requested_id)
2:      node ← random_inner_node()
3:      if not is_qualified(node) then
4:          return
5:      end if
6:      start_index ← random_index(node)
7:      i ← 0
8:      space ← 0
9:      while i < max_nodes and space < node_size do
10:         space ← space + node.child[start_index + i].free_space
11:         i ← i + 1
12:     end while
13:     if space >= node_size then
14:         merge_children(start_index, start_index + i)
15:         load_page(node.child[start_index], requested_id)
16:     end if
17: end procedure
```

Fig. 5. X-Merge Procedure

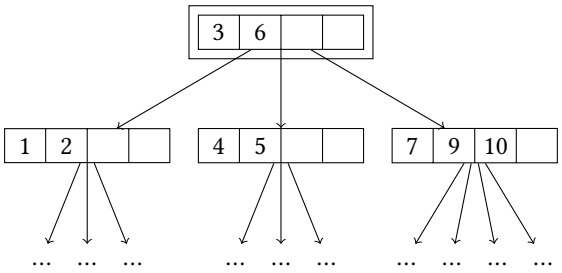

Fig. 6. Exemplary Situation before X-Merge

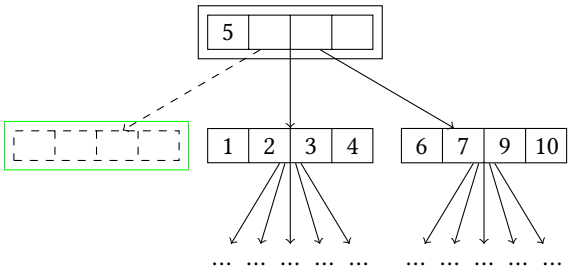

Fig. 7. Situation after X-Merge

4.1 Contention Split

Since Contention Split influences the update operation in our system, we used an update-heavy YCSB workload of 20M reads and 80M updates working on 10M preloaded entries. To have a higher chance of encountering contention on nodes, we use the Zipfian Distribution, which is preconfigured in YCSB, for the key selection of the operations. Finally, we use a *sample_prob* of 0.5% and a *period_prob* of 0.05% which we found to be good values regarding this specific environment. Because contention split works on page nodes that are in memory, we made the buffer big enough to hold all pages in main memory for this test in order to assess the maximal potential of Contention Split.

Figure 8 shows the result of running the workload against different numbers of threads performing the operations on the system with Contention Split enabled and disabled, respectively. In the beginning, we can observe an expected performance scaling. After using more than 5 threads, the positive influence of more threads starts to decrease. Using more than 10 threads reduces the performance when Contention Split is disabled due to contention occurring on the different nodes. This effect is especially noticeable after 12 threads when the number of software threads exceeds the number of hardware threads. While Contention Split cannot keep up the theoretical

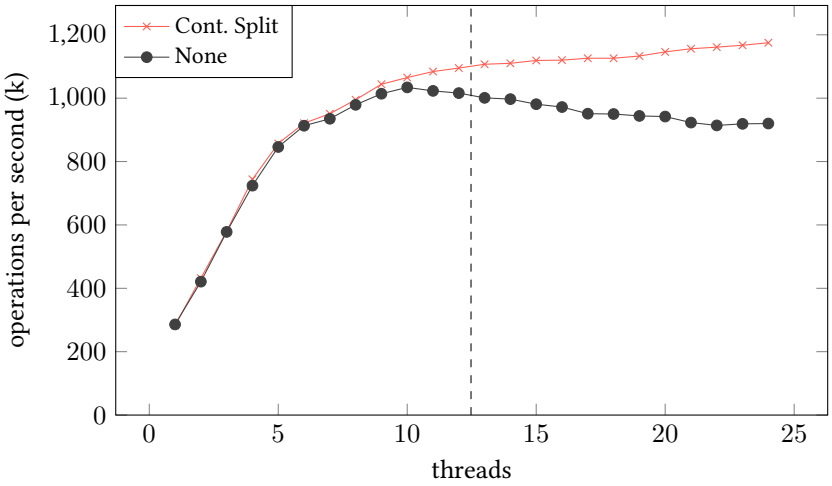

Fig. 8. Evaluation of Contention Split

performance scaling due to its limited, heuristic nature, it helps keeping the performance growth by detecting and splitting nodes that are affected by contention and create a bottleneck in the operation flow. When using 24 threads for the workload, the system with Contention Split enabled manages to process ~27% more operations per second compared to the default B$^+$-Tree.

4.2 X-Merge

For X-Merge, we adjusted the previous setup since keeping all pages in memory would completely prevent page evictions. We ran our test in three different scenarios where the page buffer would hold 75k, 37.5k, and 15k pages in memory, or about 50%, 25% and 10% of the tree nodes if we assume an average node fill rate of 66%[1]. Because X-Merge works on all B-Tree operations, we modified the workload to contain 30M reads, 60M updates, and additionally 10M inserts. At last, we set *max_nodes* to 5 and made 10 threads process the operations.
Figure 9 shows that X-Merge decreases page evictions by making space in the B$^+$-Tree for requested pages. In comparison to the system with X-Merge disabled, we can see that X-Merge increases the throughput in all three scenarios by ~8% on average. Interestingly, in scenario 3, X-Merge actually performs fewer merges than in scenario 2, although more page evictions are happening. Since X-Merge depends on eligible nodes in memory, the probability of encountering those nodes decreases with a smaller page buffer. Thus, in a scenario where only 10% of the data structure fits into memory, X-Merge succeeds more infrequently.

4.3 Contention Split and X-Merge Combined

Finally, we ran the second workload with 30M reads, 60M updates and 10M inserts for the system with both Contention Split and X-Merge enabled. We used the same parameters as in section 4.1 for Contention Split and section 4.2 for X-Merge. The results are given in Figure 9 as well.
Compared to X-Merge alone, the combination of the two techniques achieves similar results with an average difference of ~1.9%. However, it is notable that, in contrast to scenario 1 with 75k pages in memory, X-Merge alone performs better in scenarios 2 and 3 with fewer merges than in combination with Contention Split. Due to the heuristic procedure of Contention Split, the longer a page is kept in memory, the bigger the performance-optimizing effect after the respective node has been split. This means that Contention Split works best when the

[1]In general, B-Tree nodes, except for the root node, are split when completely full or merged when half full.

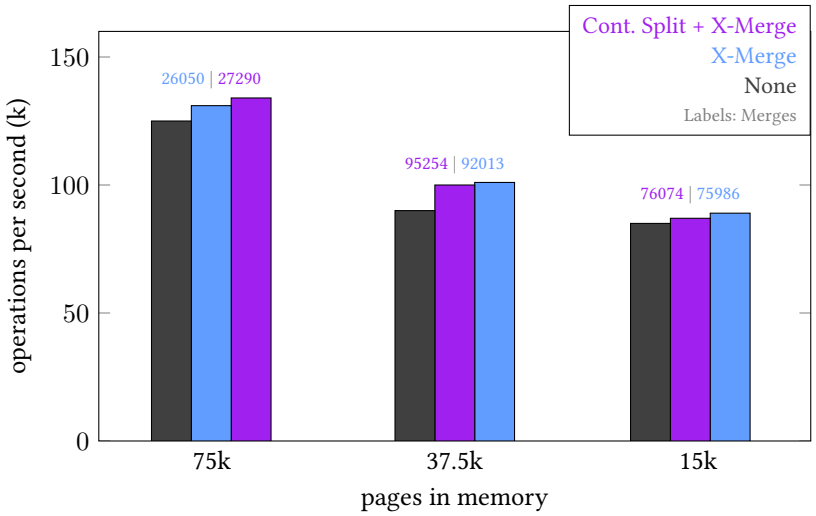

Fig. 9. Evaluation of X-Merge and the Combination of both Techniques

index is kept in memory in contrast to X-Merge which profits from a full page buffer. Indicated by the larger amount of merges in all three scenarios, X-Merge actually makes use of those nodes which have been previously split by Contention Split.

Although both techniques thus seem to work against each other procedure-wise, scenario 1 shows that the combined performance gain achieved by Contention Split working in memory and X-Merge preventing page evictions is larger than X-Merge alone. In scenarios 2 and 3, not enough nodes are kept in memory and thus the overhead caused by the additional splits and subsequent merges cannot be compensated by the effect of Contention Split anymore.

It has to be noted however, that this behavior heavily depends on the database system being used as well as the parameters for Contention Split and X-Merge. Alhomssi and Leis evaluated their implementation of Contention Split and X-Merge in a B$^+$-Tree in Leanstore [4], an OLTP storage engine, against TPC-C with a page buffer of 240 GiB and 120 worker threads (Figure 10). In addition, the parameters for Contention Split and X-Merge were explicitly evaluated and selected with the goal of maximizing the performance when using both techniques in combination.

Regarding the total throughput of the operations, the difference between Contention Split with X-Merge and Contention Split alone with the working set kept in the buffer is negligible. The same holds with X-Merge when going out of memory. Compared to the B$^+$-Tree without the two techniques, however, the combination of Contention Split and X-Merge achieve up to ~4.7x more transactions per second in memory and up to ~2.5x out of memory.

5 CONCLUSION

We can use both Contention Split and X-Merge to optimize database system usage of B-Trees. Contention Split splits nodes to prevent future contention caused by multiple threads while X-Merge preempts page evictions by merging nodes and using the freed space for requested pages. Although their strategies seem opposing regarding their effect on the data structure, the combination of both approaches outperforms a simple B$^+$-Tree as well as Contention Split and X-Merge alone. Finally, we can easily implement them into preexisting buffered environments if the page buffer has direct access to the data structure in memory.

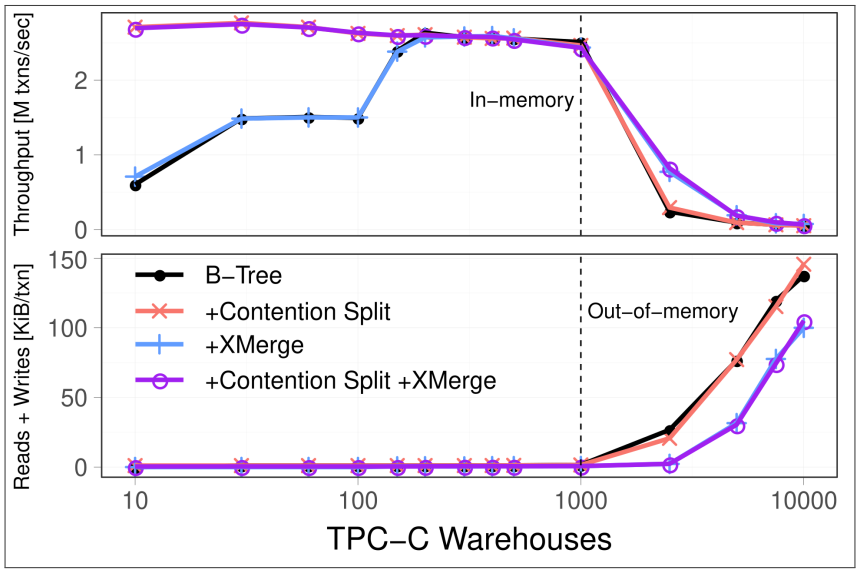

Fig. 10. Evaluation of Contention Split and X-Merge in Leanstore (Alhomssi & Leis)

REFERENCES

[1] Adnan Alhomssi and Viktor Leis. 2021. Contention and Space Management in B-Trees. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2021/papers/cidr2021_paper21.pdf

[2] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[3] Thomas Kissinger, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner. 2012. KISS-Tree: Smart Latch-Free in-Memory Indexing on Modern Architectures. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware (DaMoN '12)*. Association for Computing Machinery, New York, NY, USA, 16–23. https://doi.org/10.1145/2236584.2236587

[4] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. *2018 IEEE 34th International Conference on Data Engineering (ICDE)* (2018), 185–196.

[5] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42 (2019), 73–84.

[6] Jinglei Ren, Chris Kjellqvist, and Youngjae Lee. 2014. YCSB-cpp. https://github.com/ls4154/YCSB-cpp.

[7] Emmett Witchel. 2009. Page Replacement Algorithms. https://www.cs.utexas.edu/users/witchel/372/lectures/16.PageReplacementAlgos.pdf