Implementierungstechniken für Hauptspeicherdatenbanksysteme: The Bw-Tree

Josef Schmeißer

January 9, 2018

Abstract

The Bw-Tree as presented by Levandoski et al. was designed to accommodate the emergence of new hardware, and especially the rising number of processing cores available in modern CPU architectures. This paper provides a detailed description of the structure and algorithms of this particular tree. Another aspect of this paper concerns a performance evaluation of the Bw-Tree in comparison to B^+-Trees harnessing different locking protocols. Substantial improvements in regard to the overall performance of the B^+-Tree are to be expected by the utilization of optimistic locking techniques.

1 Introduction

Efficient synchronization techniques in the context of data structures are a fundamental aspect of achieving satisfactory scalability on modern hardware. Database index structures like B-Trees are one example where the choice of an insufficient approach might lead to an overall performance degradation. It is therefore not surprising that this field is still subject of current research.

B-Trees are traditionally often synchronized by utilizing the so called lock coupling protocol, where each page is protected by its own lock. Operations like the insertion of a new value acquire the lock on a node and keep it until it is no longer required. This is the case when the operation on the B-Tree can be sure that it will not access a certain node again. Locking goes hand in hand, this means that an operation will acquire the lock of a child node before it releases the lock of its parent node. This scheme is repeated until the desired leaf node is reached.

The lock coupling approach might seem sufficient at first glance. However, we will show that plain vanilla application of this protocol does not result in acceptable scalability. This paper on the other hand presents a different approach introduced by Levandoski et al. based on lock free techniques [1]. In fact, the presented tree does not utilize any locks at all and does therefore belong to the category of lock-free [6] data structures.

2 Related Work

Another approach was described by Lehman and Yao, the B^{link}-Tree. This particular tree utilizes a more relaxed locking scheme, where no read-locks are required. The B^{link}-Tree extends each node by a side link to its next sibling in order to maintain structural integrity. These side links will be used to redirect concurrent operations during an unfinished split. Locks are only acquired for structural operations like splits. Thereby, the B^{link}-Tree only keeps a maximum of three locks simultaneously [3]. The Bw-Tree borrows this idea of side links and utilizes the concept in a similar manner within the context of a lock-free data structure [1].

With *optimistic lock coupling* Leis et al. presented a general approach to improve scalability of lock coupling based data structures. This approach is based on the assumption that concurrent modifications are rare. Versioned locks are used to detect concurrent modifications and restart conflicting operations if necessary. This approach significantly reduces the number of writes to shared memory regions [4]. By applying this protocol to the Adaptive Radix Tree (ART) Leis et al. showed that this approach may lead to dramatic performance improvements.

3 Fundamentals

3.1 Atomic Primitives

Instead of locks the Bw-Tree makes heavy use of the so called compare-and-swap (*CAS*) instruction, which atomically replaces a memory value with a new one if the old value matches the expected value. On x86 the *CAS* instruction is implemented as `CMPXCHG` in conjunction with the `LOCK` prefix.

```
bool compare_and_swap(int * ptr, int & expected, int desired) {
    int oldValue;
    atomic {
        oldValue = *ptr;
        if (oldValue == expected) {
            *ptr = desired;
            return true;
        }
    }
    expected = oldValue;
    return false;
}
```

Listing 1: Semantics of the *CAS* instruction.

The semantics of *CAS* is shown by the pseudo-code in Listing 1. Notice, the `atomic` block which indicates that the entire block has to be performed atomically. It should be mentioned that such a feature is currently not part of C++17. There are, however, efforts to add a similar feature to the language standard [2].

4 Architecture

The Bw-Tree is a lock free variant of the B$^+$-Tree, payload is hence only stored on leaf nodes. It is mainly based on one important key idea: updates are performed in a lock-free manner through delta records. Such a delta record is considered to be immutable once it is part of the tree. We never alter the existing state, every change to the tree is formulated as delta, and will be prepended to a chain of existing records [1]. The base page is also part of this chain. We will refer to the whole record chain (which includes the base page) simply as page or sometimes as logical page. Structural modifications like splits can be formulated with multiple delta updates. This will be described in more detail in the following sections.

Another key idea is the use of a so called Mapping Table. This table can be implemented as an array of pointers to records. We will refer to an index of this table as Page Identifier or simply as PID. Hence, this table represents a mapping function of the form $PID \rightarrow ptr$. Changes to this table are usually performed by an atomic CAS operation. Conflicting operations to a single page are therefore easy to detect by the result of the CAS [1]. We usually repeat a failed operation simply by restarting it if the CAS fails.

4.1 Leaf Level Modifications

Most of the time we perform update operations on a leaf page. There we use three types of delta records: (1) *insert*, which represents a new key-value tuple; (2) *modify*, which represents a modification to an existing entry; and (3) *delete*, which represents the logical deletion of an existing entry [1]. These updates may, however, lead to more complex structural modifications like a split, which will be described subsection 4.3.

Searching a key on a leaf page is more involved than in a normal B$^+$-Tree. We start with the first delta record (if present) and traverse the delta chain until we find a delta which contains the key, or until we have reached the base page. The search succeeds if the key is contained in either an *insert* or *update* delta and fails if it is part a *delete* delta. Otherwise, we perform a binary search on the base page in the standard B$^+$-Tree manner [1].

4.2 Page Consolidation

Constantly appending deltas to logical pages leads to ever-expanding chains, which in turn leads to poor search performance [1]. We thus consolidate pages from time to time, once a certain chain length threshold is reached. A page is consolidated by creating a copy of the existing base page, and applying all delta changes to this copy. Afterwards, the consolidated page will be installed with a CAS instruction. The memory of the old logical page will be reclaimed once the page is no longer used.

4.3 Node Split

Splitting a node requires additional effort within the context of a Bw-Tree. The split logic basically consists of two parts: (1) splitting the corresponding page and (2) altering the parent node. Both parts have to be performed atomically [1].

```
struct LeafSplitDeltaRecord : public Record {
    const Key separatorKey;
    const pid_ty upper;
};
```

Listing 2: `LeafSplitDeltaRecord` holds all attributes associated with a certain page split. A similar record also exists for inner nodes.

In order to split page P we have to determine an appropriate separator key k_s. This key is used to consolidate every entry with key k_i which satisfies $k_s > k_i$ into a new page P_r. The newly created page P_r is then installed into the mapping table. Note that we do not require an atomic CAS operation to accomplish this since only the thread performing the split is aware of P_r at this time [1]. P_r and k_s are then collected in an instance of `LeafSplitDeltaRecord`, which is shown in Listing 2. We will then try to install the newly created record into the mapping table via a CAS instruction. If the split fails at this point, we have to deallocate the memory previously allocated for P_r and invalidate the according PID. Another thread or a future operation may initiate a new attempt to split P. If, however, installing this record succeeds, the first part of the split is considered to be complete.

It is important to note that the separator key k_s stored in the `separatorKey` field also acts as an invalidation hint. Every entry with key k_i within P, for which $k_s > k_i$ holds, will from now on be considered as obsolete since it now resides on page P_r. A subsequent consolidation process will therefore only consolidate entries which satisfy $k_i \leq k_s$[1]. The `upper` field on the other hand contains the side pointer (which is a PID) to P_r. This is necessary to ensure structural integrity during complex operations (like a split). To see this, consider a situation where only the first part of a split is complete. Since we did not yet install our separator k_s in the parent, a tree traversal for a key k_t with $k_t > k_s$ will initially end up on page P. However, it must yield P_r as result in order to retain our invariant. This will be achieved by following the side link to P_r within the split record while processing the record chain of P.

```
struct InnerIndexEntryRecord : public Record {
    const pid_ty child = invalidPid;
    // a record of this type represents the range: (lowKey, highKey]
    const Key lowKey;
    const Key highKey;
    // or the range (lowKey, ∞) if 'hasUpperBound' is not set
    const bool hasUpperBound;
};
```

Listing 3: `InnerIndexEntryRecord` extends an inner node by one entry.

In the second part of the split we update the parent node P_p of P. Thus, we create an instance of `InnerIndexEntryRecord`, which is shown in Listing 3. This struct holds the child PID namely the PID of P together with two key entries. These keys represent the responsibility domain of P, hence `lowKey` will be set to k_s whereas `highKey` will be set to the following separator key on P_p if such a key exists, otherwise `lowKey` will be left empty. This is the case when P is the rightmost child of P_p, this scenario will be indicated by setting `hasUpperBound` to `false`. Note that the domain of P is unlimited in regard to the upper limit in the latter case. After the initialization of these members we will attempt to install the resulting record by an atomic CAS operation as usual. This, however, may fail. Recovering from this situation is more involved than usual.

Let us first consider how P_p is determined in the first place. During the initial tree traversal we remember the full path of $PIDs$ which brought us to P. The direct ancestor P_p of P is therefore also part of this path. However, occasionally it may occur that the parent P_p has been logically deleted in the meantime. We are able to detect this situation by the presence of a *remove node delta* record indicating that P_p is no longer part of the tree structure. In this case we have to determine the new parent of P by re-traversing the tree.

P_p's size might exceed the page size limit after successfully installing the newly created *index entry record* instance. If this is the case, we have to recursively repeat the split process and apply it to P_p as well. Occasionally it may happen that a recursive split propagates up to the root node. In this case we have to increase the tree depth by creating a new root node. We then try to install the newly created root node with an atomic CAS operation and repeat the previous steps if the atomic CAS fails.

4.4 Node Merge

Once the logical size of page is below a certain threshold, a node merge is initiated. However, we need even more atomic operations than in the split case to perform the merge operation in a lock-free manner [1].

We first mark a page P_r as obsolete by installing a *remove node delta* record [1]. The only purpose of this record is to act as a marker, it has no further attributes. This ensures that other threads will not attempt to access P_r. The *remove node delta* record indicates that the left sibling P_l of P_r is now responsible for the combined domain of both nodes, it thus also acts as redirection to the left sibling. A thread encountering such a record on page P_r has to perform its operation on P_l instead.

```
struct LeafNodeMergeRecord : public Record {
    const Key separatorKey;
    const Record * rightPage;
};
```

Listing 4: `LeafNodeMergeRecord` holds all attributes associated with a certain merge. A similar record also exists for inner nodes.

The second part deals with the actual merge of P_l and P_r. We perform the merge by posting a *node merge delta* record such as shown in Listing 4 to P_l. Note that we use a physical pointer as reference to P_r in this case. The `rightPage` field points to the first record after the *remove node delta* record on P_r. This indicates that the contents of P_r are to be included in P_l [1].

Including the separator key k_s in the field `separatorKey` as shown in Listing 4 serves an important purpose when searching P_l for keys contained in P_r's domain. A thread searching P_l will be either directed to P_l's original state or to the absorbed contents of P_r (these are still accessible through the `rightPage` pointer) [1]. The tree like structure of this record chain will be resolved during page consolidation. Consolidation on P_l is triggered as usual when the system notices that the chain length has grown beyond a certain threshold.

In the third and final part of the merge we have to alter the parent node P_p of P_r by removing the corresponding index entry, which still points to P_r. The index entry will be logically removed by posting an *index term delete* delta record. This record contains the information that P_r is from now on considered to be deleted and the new key domain of P_l. We specify the new domain as union of P_l's and P_r's domain by choosing the low key of P_l and the high key of P_r as key range within the *index term delete* delta [1]. Subsequent searches for a key in P_r's domain are thus redirected to P_l.

It may be necessary to recursively merge the parent node P_p as well. The applied logic is similar to the recursive split, thus the merge operation may also propagate up to the root node.

5 Alternative Approach

As already mentioned, the optimistic lock coupling protocol can be used as an alternative to the normal lock coupling protocol. Converting a B$^+$-Tree implementation based on lock coupling to the optimistic lock coupling protocol proved to be fairly simple, only minor adoptions were necessary. These adaptations were limited to additional version verifications as described by Leis et al. [4]. For example, one has to ensure that a child pointer is still valid during a tree traversal. Furthermore, it is necessary to ensure that every optimistically operating algorithm terminates, since a concurrent operation on the same node may perform arbitrary changes.

6 Performance Evaluation

The subsequent experiments were conducted on the following test system: Intel® Core™ i9-7900X with a total amount 10 processing cores hyper-threaded to 20 logical cores.

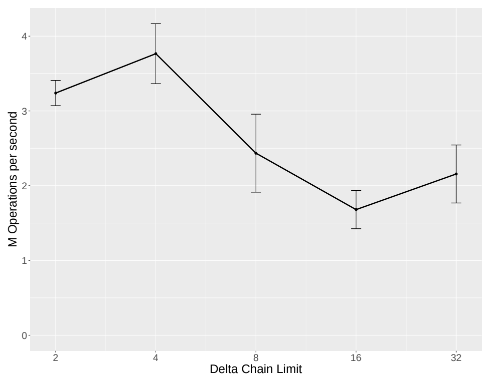
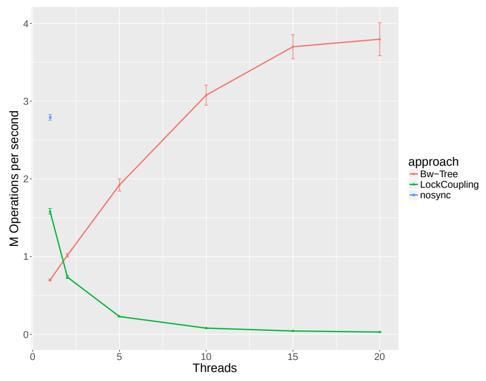
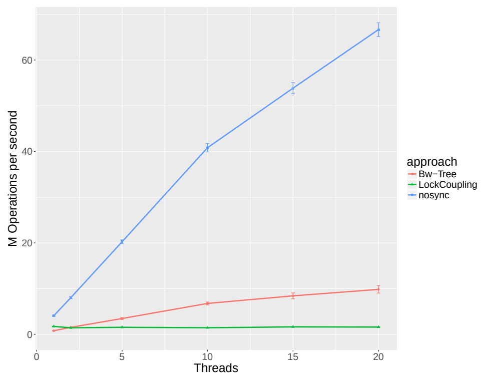


Figure 1: Performance effect of varying chain length limits.

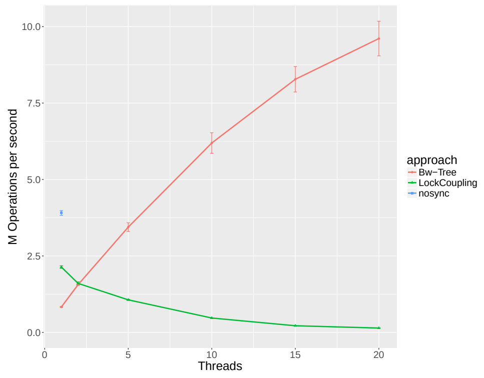
The effect of varying delta chain length thresholds is depicted in Figure 1. Our synthetic workload confirmed the results shown by Levandoski et al. [1]. Shorter delta chain length thresholds lead to more frequent page consolidations, which in turn results in a performance degradation; whereas, on the other hand, larger thresholds also lead to a significant performance decline once a certain limit is transcended. The optimal choice for this threshold depends on the workload as shown by Levandoski et al. [1]. In our case a limit of four seems to be the optimal choice for the threshold.



(a) Insert Workload



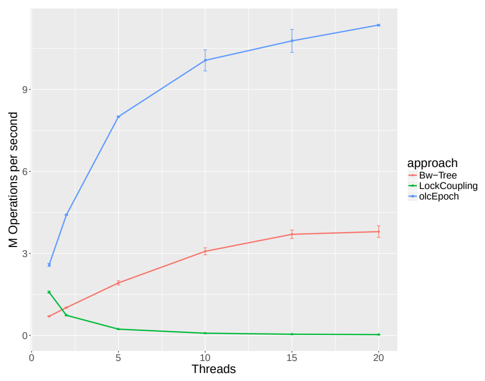
(b) Lookup Workload



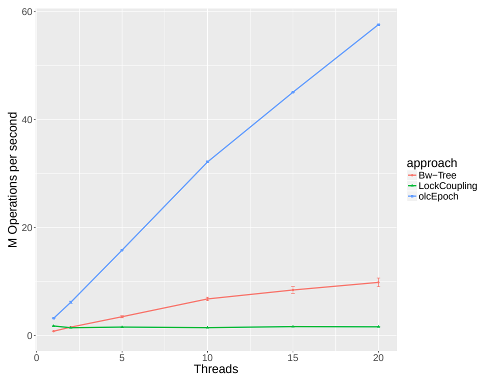
(c) Mixed Workload

Figure 2: Performance evaluation in comparison to Lock Coupling.

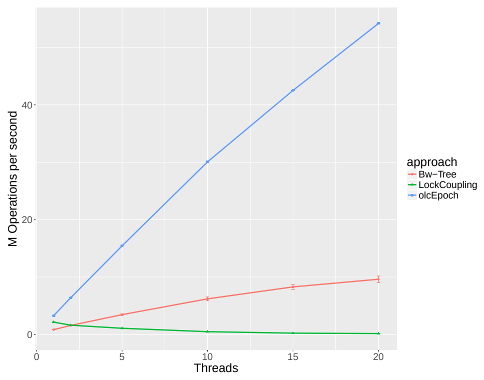
Levandoski et al. compare their implementation against BerkeleyDB's B-Tree in their analysis [1]. This particular B-Tree implementation utilizes the lock coupling synchronization protocol [5]. Their analysis yielded a significant performance increase in favor of the Bw-Tree. Our implementation was able to confirm this advantage over the lock coupling based approach as shown in Figure 2.



(a) Insert Workload



(b) Lookup Workload



(c) Mixed Workload

Figure 3: Performance evaluation in comparison to Optimistic Lock Coupling.

However, the approach described in section 5 yields quite a different picture as shown in Figure 3. The optimistic lock coupling (olc) based approach in conjunction with an epoch-based memory reclamation scheme as described by Fraser [7] results in far better scalability throughout the entire collection of conducted benchmarks.

7 Conclusion

When considering scalability, the Bw-Tree is definitely an improvement over the traditional lock coupling based approach. However, more sophisticated locking protocols such as optimistic lock coupling yield far better results in terms of scalability. Another disadvantage of the Bw-Tree is the rather poor single-threaded performance revealed by the previously shown experiments.

References

[1] Justin J. Levandoski, David B. Lomet and Sudipta Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. IEEE 29th International Conference on Data Engineering (ICDE), 2013.

[2] ISO/IEC JTC 1/SC 22/WG 21. Technical Specification for C++ Extensions for Transactional Memory. ISO, 2015.

[3] Philip L. Lehman and S. Bing Yao. Efficient Locking for Concurrent Operations on B-Trees. ACM Transactions on Database Systems, Vol. 6, No. 4, December 1981, Pages 650-670.

[4] Viktor Leis, Florian Scheibner, Alfons Kemper and Thomas Neumann. The ART of Practical Synchronization. Twelfth International Workshop on Data Management on New Hardware, 2016.

[5] Margo Seltzer and Keith Bostic. Berkeley DB. http://www.aosabook.org/en/bdb.html, accessed: 28.12.2017.

[6] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. Java Concurrency in Practice. Addison-Wesley, 2005.

[7] Keir Fraser. Technical Report: Practical lock-freedom. University of Cambridge, 2004.