Seminar Advanced Algorithms and Data Structures - Student Report

Write-Optimized Skip Lists

Fabian Grob

Introduction

This report is about the paper *Write-Optimized Skip Lists* [1] written by Michael A. Bender, Martìn Farach-Colton, Rob Johnson, Simon Mauras, Tyler Mayer, Cynthia A. Philips and Helen Xu which presents an external-memory skip lists that achieves write-optimized bounds.
The regular skip list with N elements supports searches, inserts and deletes in $\mathcal{O}(\log N)$ operations with high probability[1] (w.h.p.), range queries returning K elements can be done in $\mathcal{O}(\log N + K)$ operations w.h.p. It is an elegant dictionary data structure that is commonly deployed in RAM.
There seems to be a natural way to generalize the skip list for external-memory with block size B: Promote[2] with probability $1/B$ instead of $1/2$. This basically would work but the efficient performance, space bounds and high-probability guarantees can not be hold. More often the B-tree is used which offers the for searches, inserts and deletes with $\mathcal{O}(\log N)$ operations the same bound as the skip list for internal memory.
The paper proposes a write-optimized skip list, which is a randomized external-memory directory. The proposed write-optimized skip list offers an asymptotically better performance in insertions and deletions than a B-tree whereas the query performance stays near the speed of the B-tree. Detailed numbers are found in Table 1, as usual for ϵ holds $0 < \epsilon < 1$.

	Insert and Delete	Query
B-Tree	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$
Write-Optimized Skip List	$\mathcal{O}(\log_{B^\epsilon} N/B^{1-\epsilon})$	$\mathcal{O}(\log_{B^\epsilon} N)$

Table 1: Performance B-Tree and write-optimized skip list (w.h.p.)

This report focuses on the structure of the write-optimized skip list and its insertions. Queries and deletions are mentioned briefly but not explained in detail, but they are explained in the original paper [1].

External memory model

In computing, the external memory model is used if the processed data structure is too large to fit into the main memory. This data structure lies on a large external storage device such as a disk or a SSD. This device is accessed via I/Os that transfers blocks of size B to a smaller cache (e.g. RAM) of size M.

[1] An event E_n on a problem of size n occurs *with high probability* if $P[E_n] \geq 1 - 1/n^c$ for some constant c

[2] This means, an element is pushed to a higher level than its internal height.

Skip list

A skip lists with N elements has $h = \mathcal{O}(\log N)$ levels $\{\mathcal{L}_0, \mathcal{L}_1, ..., \mathcal{L}_h\}$, each level is a sorted linked list, the base level $\mathcal{L}_0$ contains all items. An item of a level $\mathcal{L}_i$ can be promoted to a the next level $\mathcal{L}_{i+1}$ with probability $1/2$, this means they appear also in $\mathcal{L}_{i+1}$ and have an additional pointer to its "source"-item in the level $\mathcal{L}_i$. Additional to the elements each level contains also a base item $-\infty$ which symbolizes the start of the linked list.
Each query for an element starts at the smallest element in level $\mathcal{L}_h$. If the next element in the current level is lower than the current element the query follows inside the level (horizontal pointers in figure 1), otherwise follow the pointer to the next level (vertical pointers in figure 1 horizontal).
For insertions the element is first inserted into $\mathcal{L}_0$. Afterwards a coin is tossed until the first tail occurs but maximal $h + 1$ times or until. Each times a head occurs the element gets promoted with appropriate adjustments of all pointers. For deletions the element is deleted in all levels. If afterwards level $\mathcal{L}_h$ is empty, remove it.

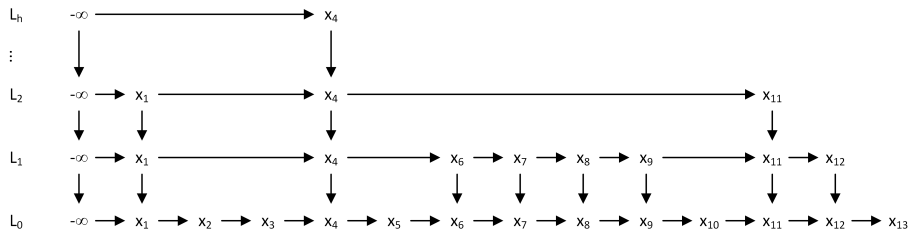

Figure 1: Example of a Skip-List

B-tree

For external-memory the Skip list is not optimal, mostly B-trees are used. B-trees are somewhat similar to Skip lists as they both are hierarchical with $\mathcal{O}(\log N)$ levels. B-trees consist of multiple lists containing values. Between two values v_1, v_2 there is a pointer to another list containing just values between v_1 and v_2 (with "invisible" values $-\infty$ and ∞ at the border of each list). Figure 2 does not show pointers to empty lists. So for each element there is just one possible path and not every element has a corresponding leaf. In comparison to Skip list, B-trees are balanced so there is no randomness involved. It has the same bounds for queries, inserts and deletions but these bounds are fixed and not with high probability.

Write-optimized skip list

The basic idea for the write-optimized skip list is to link nodes instead of elements. These nodes have size $\Theta(B)$ w.h.p. and are filled with pivots (elements pointing to another node) and a buffer of elements. Each node has at least one pivot. The smallest pivot of a node is called leader. If an element is a pivot in $\mathcal{L}_i$ it has a pointer to its corresponding pivot in $\mathcal{L}_i - 1$. The corresponding pivot has to be leader of his node.

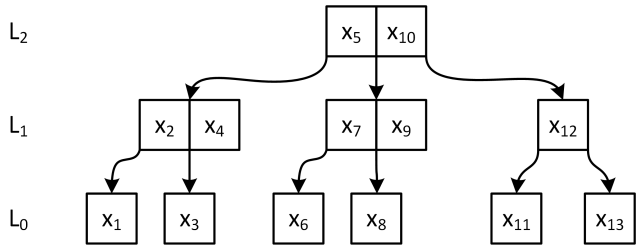

Figure 2: B-Tree

The buffer of a node is filled with elements. Those elements are ordered and all of them have to be greater than the leader and smaller than the leader of the next node of the same level. If one inserts an element, it is first stored in the buffer. As soon as the buffer is full, it is flushed in batches from parents to children. Thus, all elements are first insert into the root node at level $\mathcal{L}_h$ and move towards $\mathcal{L}_0$ where they remain unless they are deleted. This means that the performance depends on the cost and frequency of such flushes.
Lets look at the node in level 2 of Figure 3: $-\infty$ is the leader of the node, 16 and 42 are pivots and 7, 9 and 53 are in the buffer.

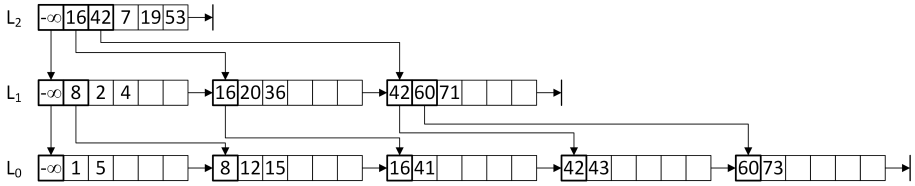

Figure 3: Example of a Write-Optimized Skip-List with block size $B = 6$

Randomized balancing

Each element e has an integer height h_e determined by a sequence of biased coin flips. Coin flips are implemented by hashing e so it does not change even if an element is inserted, deleted or changed. To determine h_e a biased coin is flipped until the first tail, h_e is then set to the number of heads. There are two different coins, one for the first flip with heads-probability of $1/B^{1-\epsilon}$ and one for the following flips with heads-probability $1/B^{\epsilon}$. An element e has been *promoted* to level $i > 0$ if $h_e \geq i$. For all the report, $0 < \epsilon < 1$ holds.
Intuitively the height says below which level the element becomes a pivot when flushed out of the buffer, a more detailed explanation is in the next section.
To ensure that the structure contains a root, there is a special element $-\infty$ that is defined to have the largest height of any element.

Buffer flushing

When a buffer in node D at level $\mathcal{L}_{i\geq 1}$ overflows, a *flush* operation is needed (Figure 4). All elements of D's buffer are distributed among D's children, so the elements of the buffer are "flushed" one level deeper.

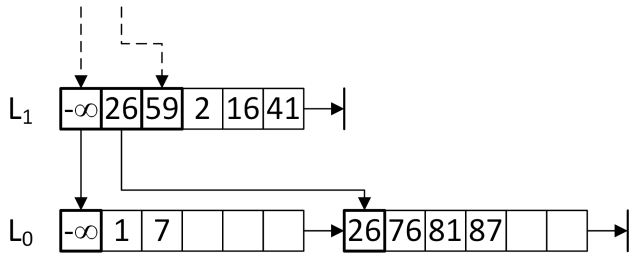

Figure 4: We want to add 43 to the node at level 1, so it overflows.

If the level i of the previous node D is smaller or equal than the height h_e of the flushed element e ($i \leq h_e$), e becomes a pivot of D in addition of getting flushed (Figure 5). So afterwards D has multiple pivots pointing to the same child.

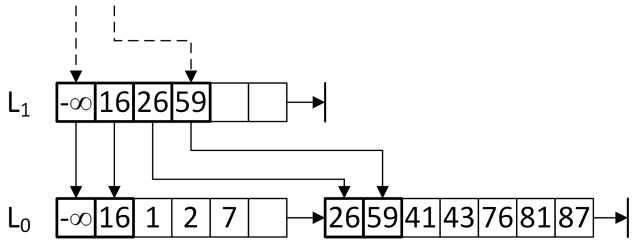

Figure 5: Heights: $h_{16} = h_{26} = 1$, $h_{59} = 2$, the other elements have height 0

If the height h_e is strictly smaller ($i < h_e$), then split D into two nodes D' and D''. The current leader of D will be the leader of D', e becomes the leader of D''. Update all the parents of D to point to the newly created nodes (Figure 6). We have $i < h_e$ with all integers, thus e must be already a pivot in the parent node because $i \leq h_e$ holds for the parent node, so the size of the parent remains unchanged.

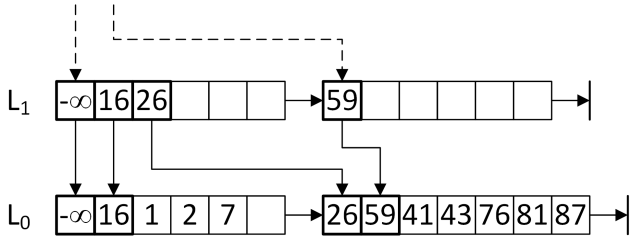

Figure 6: The node at level 1 is split because $h_{59} = 2$

Leaves (nodes at level 0) require special handling. When a node D at level 1 flushes elements to the leaves, rebalance all the leaves by greedily choose the

breaks between leaves so that each leaf approximately fills a block and each leaf begins with a pivot of D (Figure 7).

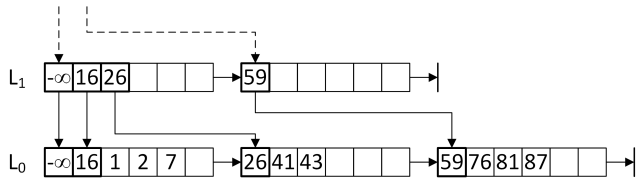

Figure 7: The leaf nodes are now balanced.

Structural Bounds

To analyse the performance, a few bounds of the structure are needed.
We assume that $\min(B^\epsilon, B^{1-\epsilon}) \geq \log N$.

Pivots in an Internal Node [Lemma 2]. *An internal node has B^ϵ pivots in expectation and $\mathcal{O}(B^\epsilon \log N) = \mathcal{O}(B)$ pivots w.h.p.*

Proof. By construction new internal nodes are created when we see a promotion to the next level. So the number of pivots in each internal nodes can be modeled as the number X of tails before the first head in a sequence of independent coin flips with a head probability of $B^{-\epsilon}$, this leads to $E(X) = B^\epsilon$. The high probability bounds follow from the Chernoff bounds. □

Node Size [Lemma 3]. *For $0 < \epsilon < 1$, a node fits into $\mathcal{O}(1)$ blocks w.h.p.*

Proof. Nodes in levels greater than 0 contain pivots and $\Theta(B)$ buffer space. By Lemma 2, nodes have $\mathcal{O}(B)$ pivots w.h.p., so a node fits into $\mathcal{O}(1)$ blocks w.h.p. Although the promotion probability at the leaves (nodes at level 0) is $1/B^{1-\epsilon}$, by the same argument every run of $\Theta(B)$ elements at level 0 has a promoted element w.h.p. So when the elements at level 0 are packed into blocks, a new leaf can be created every $\Theta(B)$ blocks w.h.p. thus every node at level 0 fits in $\mathcal{O}(1)$ blocks w.h.p. □

Neighbor bounds [Lemma 4]. *Let D be a node at height at least 1, the number of its parents is $\mathcal{O}(1)$ w.h.p. and the number of its children is $\mathcal{O}(B^\epsilon)$. If the height is exactly 1, then D has $\mathcal{O}(B) = \mathcal{O}(B^\epsilon \log N)$ children w.h.p.*

Proof. If D is at level $i > 1$ then, by Lemma 2, we expect $\mathcal{O}(B^\epsilon)$ pivots and therefore we also expect $\mathcal{O}(B^\epsilon)$ children.
Nodes at level 1 are split whenever an element is promoted to level 2. Elements in level 0 have a $1/B$ chance of being promoted to level 2. Any run of $\Omega(B \log N)$ elements at level 0 has at least 1 element promoted to level 2 w.h.p. Thus, w.h.p. no node at level 1 has more than $\mathcal{O}(B \log N)$ elements in its children. Since each child has $\Theta(B)$ elements, nodes at level 1 have $\mathcal{O}(B) = \mathcal{O}(B^\epsilon \log N)$ children w.h.p.
The number of parents of D is at most the number of elements in D's buffer that have height at least 2 larger than the height of D. Since D has height at least 1, the probability that any particular item in D's buffer has height 2 greater than the height of D is at most $\mathcal{O}(1/B^{1+\epsilon})$. Since D's buffer contains

$\mathcal{O}(B)$ items, the expected number of such elements in D's buffer is $\mathcal{O}(1/B^\epsilon)$. Thus, the number of such elements is $\mathcal{O}((\log N)/B^\epsilon)$ w.h.p. Since $\log N < B^\epsilon$, the number of such elements, and hence the number of parents of D is $\mathcal{O}(1)$. □

Height Upper Bound [Lemma 6]. *For $0 < \epsilon < 1$, the height of the write-optimized skip list is $\mathcal{O}(\log_{B^\epsilon} N)$ in expectation and w.h.p.*

Proof. The probability that any given element has height at least $h \geq 1$ is $1/B^{1-\epsilon+(h-1)\epsilon} = 1/B^{1+(h-2)\epsilon}$.
Let $c \geq 2$ be a constant. The probability that any given element has height at least $h = 1 + c\log_{B^\epsilon} N$ is at most

$$1/B^{1+(h-2)\epsilon} \leq 1/B^{\epsilon(h-1)} = 1/B^{\epsilon c \log_{B^\epsilon} N}$$

The probability that any given element has height at least $1 + c\log_{B^\epsilon} N$ is at most $1/N^c$. By the union bound, the probability that any of the N elements has height at least $1 + c\log_{B^\epsilon} N$ is at most $1/N^{c-1}$. □

Performance analysis

As the skip list is write-optimized, the write-analysis is the most interesting performance. For writes we look at the insert-operation. During an insert, the element is first added to a buffer, this can be done in $\mathcal{O}(1)$ time. Afterwards there are sometimes flushes needed, so we need to take a look to the cost of flushes.
A flush of a node D accesses all children and (during splits) parents of D. By Lemma 4, we expect a node to have $\mathcal{O}(B^\epsilon)$ parents and children, so during a flush we expect a total number of $\mathcal{O}(B^\epsilon)$ nodes to be accessed. By Lemma 3 each node fits in $\mathcal{O}(1)$ blocks w.h.p., so the total number of I/Os required by a flush is $\mathcal{O}(B^\epsilon)$.
Now we can use this to analyse the expected amortized insertion cost. From Lemma 6 we conclude that each element must be flushed $\mathcal{O}(\log_B N)$ times w.h.p. so for any sequence of N insertions we must perform $\mathcal{O}(N\log_B N)$ element-flushes w.h.p. Because each flush performs $\Theta(B)$ element-flushes w.h.p., the total number of flushes performed is $\mathcal{O}(N\log_B N/B)$. As each flush costs $\mathcal{O}(B^\epsilon)$ I/Os in expectation, the amortized insertion cost is $\mathcal{O}(\log_B N/B^{1-\epsilon})$.
Chapter 5 of the paper [1] gives a proof, that if the cache is big enough, the expected cost for insertion $\mathcal{O}(\log_B N/B^{1-\epsilon})$ holds with high probability.

Conclusion

Not explained in this report are queries and deletions. These two actions are explained in the paper [1] too, point queries have a worst case I/O complexity of $\mathcal{O}(\log_{B^\epsilon} N)$. For deletions the write-optimized skip list inserts *tombstones* which behaves differently during flushes, so deletions have the same complexity as inserts.
Regular skip lists are used for concurrent and lock-free production directories. The authors of the paper believe that the write-optimized skip list benefits from the same implementation advantages as the regular skip list. In comparison to B-trees, node splits of the write-optimized skip list are triggered "on the way down", so it is easier to implement locks.

The authors hope, that write-optimized skip lists will become the easiest-to-implement lock-free write-optimized data structure. For that they plan in further research to perform an implementation study to explore if the theoretical advantages lead to benefits for implementers and users.

References

[1] Michael A Bender et al. "Write-Optimized Skip Lists". In: *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM. 2017, pp. 69–78.