Implementation of an External-Memory Fractal Tree in the C++ STXXL Library

Henri Froese
Goethe-Universität Frankfurt am Main

January 24, 2021

Contents

1 Introduction

This report describes the implementation of an external-memory fractal tree in the C++ STXXL Library[1]. The code can be found here[2]. Section 2 provides the implementation's theoretical background. Section 3 describes in-detail the implementation. Finally, section 4 benchmarks the implementation in context.

2 Background

2.1 External-Memory Model

When working with data sets that do not fit into internal memory (RAM, main memory), a part of the data will always need be stored in external memory. External memory access is several orders of magnitude slower than internal memory access and commonly occurs in blocks, where transferring a chunk of data is only marginally more expensive than tranferring a singly byte. Thus, the efficiency of data structures for external memory is dominated by the number of disk I/Os. This motivates the external-memory model from [1].

In the model, the processor can access the internal memory of size M. In one operation, one block of size B can be read or written from unlimited external memory. The complexity is defined by the (asymptotic) number of I/Os - i.e. operations on blocks in internal memory are 'free'.

2.2 Different Trees for External Memory

Consider the problem of implementing a map data structure in external memory, mapping keys to values and supporting operations

- `insert(key, value)`
- `value = find(key)`
- `values = range_find(key_lower, key_upper)`
- `delete(key k)`

2.2.1 Binary Search Tree

Initially, one might use a balanced *Binary Search Tree* (BST) such as an AVL- or Red-Black-Tree: The tree has depth $\mathcal{O}(\log N)$ when storing N key-value pairs. Thus, the first three operations are supported in $\mathcal{O}(\log N)$ steps, where N is the number of keys in the map. The last operation is supported in $\mathcal{O}(\log(N) + X)$ steps, where X is the number of keys in the range.

However, in the worst case, for each node of the tree that is traversed, we might need one I/O to fetch the node's key and value. Thus, in the worst case the complexity of balanced BSTs is the same as the number of operations necessary. The BSTs do not profit at all from the fact that each I/O fetches a whole block of size B and instead use only one item from the block per level.

[1]https://stxxl.org/

[2]https://github.com/henrifroese/external_memory_fractal_tree

2.2.2 B-Tree

Using a *B-Tree*, a balanced search tree with B keys per inner node, allows us to fill up one block with each node and thus fully profit from the paged nature of the memory. The depth is $\mathcal{O}(\log_B(\frac{N}{B}))$, so the first three operations incur $\mathcal{O}(\log_B(\frac{N}{B}))$ I/Os. The operations necessary to find the correct key in a node are 'free' as they work on the fetched block and don't incur any additional I/Os. `range_find` is supported in $\mathcal{O}(\log_B(\frac{N}{B}) + \frac{X}{B})$ I/Os.

2.2.3 Buffered Repository Tree

A *Buffered Repository Tree (BRT)*, introduced in [2], explores a different direction than the B-Tree to extend BSTs to profit off of getting a whole block of data per I/O. Instead of increasing the branching factor from a constant to B, each node consists of a constant number of key-value pairs used for branching (henceforth the *pivots*) and fills the remainder of the block with a buffer of size ca. B. To insert an item, it is added to the root's buffer. If the root's buffer overflows, the buffer items are recursively inserted into the appropriate children nodes' buffers. A BRT has depth $\mathcal{O}(\log(\frac{N}{B}))$. Insertions incur an amortized $\mathcal{O}(\frac{1}{B}\log(\frac{N}{B}))$ I/Os (see [3]). Searches incur $\mathcal{O}(\log(\frac{N}{B}))$ I/Os, and `range_find` incurs $\mathcal{O}(\log(\frac{N}{B}) + \frac{X}{B})$ I/Os.

2.2.4 Buffered Repository Bϵ-Tree

We can see that filling a node's block with a buffer (BRTs) instead of more pivots (B-Trees) leads to faster insertion and slower search. This tradeoff between search and insertion performance is navigated in the *Buffered Repository Bϵ-Tree (BRBϵ-Tree)*, introduced in [3]. A BRBϵ-Tree splits a node's block into $\approx B^\epsilon$ pivots and $B - B^\epsilon$ buffer items. For small values of ϵ close to 0, this is a BRT, and for ϵ close to 1, this is a B-Tree. Larger values for ϵ give a greater branching factor (favoring searches) and smaller values come with a bigger buffer (favoring insertions).

The depth of a BRBϵ-Tree is $\mathcal{O}(\frac{1}{\epsilon}\log_B(\frac{N}{B}))$, giving a search performance of $\mathcal{O}(\frac{1}{\epsilon}\log_B(\frac{N}{B}))$ and an insertion performance of $\mathcal{O}(\frac{1}{\epsilon \cdot B^{1-\epsilon}}\log_B \frac{N}{B})$ (see [4]). Range searches cost $\mathcal{O}(\frac{1}{\epsilon}\log_B \frac{N}{B} + \frac{X}{N})$ I/Os. Setting ϵ to 0.5 gives interesting properties - the resulting tree is called a *Fractal Tree.* See table 1 for a summary of the complexities of the showcased trees, similar to the one in [4].

2.2.5 Fractal Tree

As we can see in table 1, the *Fractal Tree* – a Buffered Repository Bϵ-Tree with $\epsilon = 0.5$, so $\approx \sqrt{B}$ pivots and $\approx B - \sqrt{B}$ buffer items – strikes a balance between the sizes of the buffer and the pivots that preserves the advantage of the B-Tree (only pivots, no buffer, permitting

[3]It costs 1 I/O to flush a node's B buffer items to the node's constant number of children; assigning each item its share of $\frac{1}{B}$ I/Os and noting that an item is flushed at most down the whole tree with depth $\mathcal{O}(\log(\frac{N}{B}))$, gives the amortized bound.

[4]A node has B^ϵ children, so flushing a buffer of $B - B^\epsilon$ items costs B^ϵ I/Os. Amortizing the I/Os over the items to $\frac{B^\epsilon}{B-B^\epsilon}$ per item and again noting that an item is flushed at most down the whole tree, gives the amortized bound.

Type	Insert	Search	Range-Search
BST	$\log(N)$	$\log(N)$	$\log(N) + X$
B-Tree	$\log_B(\frac{N}{B})$	$\log_B(\frac{N}{B})$	$\log_B(\frac{N}{B}) + \frac{X}{B}$
BRT	$\frac{1}{B}\log(\frac{N}{B})$	$\log(\frac{N}{B})$	$\log(\frac{N}{B}) + \frac{X}{B}$
BRBϵ-Tree	$\frac{1}{\epsilon \cdot B^{1-\epsilon}}\log_B(\frac{N}{B})$	$\frac{1}{\epsilon}\log_B(\frac{N}{B})$	$\frac{1}{\epsilon}\log_B(\frac{N}{B}) + \frac{X}{B}$
Fractal Tree	$\frac{1}{\sqrt{B}}\log_B(\frac{N}{B})$	$\log_B(\frac{N}{B})$	$\log_B(\frac{N}{B}) + \frac{X}{B}$

Table 1: Asymptotic I/O complexities of presented tree data structures.

fast search) and the BRT (few pivots, mostly buffer, permitting fast insertions). Thus, the Fractal Tree makes for a compelling data structure to implement and test.

2.3 STXXL

The *Standard Template Library for Extra Large Data Sets* (STXXL) provides data structures and algorithms for external memory applications, with implementations exposing roughly the same interface as the C++ Standard Template Library (STL). An introduction and descriptions of the design and internals can be found here[5]. Additionally, here[6] is a quick introduction adapted from the official tutorials that showcases the basics to get started with the STXXL without any prior knowledge of the library.

3 Fractal Tree Implementation

Currently, everything except the delete-operation is implemented. Thus, the API is currently

- `insert(key, value)`
- `value = find(key)`
- `values = range_find(key_lower, key_upper)`

The main goals of the implementation are to make the fractal tree easily usable, well-performing with small deviations from the asymptotic complexity, and fully configurable with regards to block size, cache size, and key and data types.

Here's an example of how the tree can be used:

```cpp
using key_type = unsigned int;
using data_type = unsigned int;
constexpr unsigned block_size = 4096;
constexpr unsigned cache_size = 8 * 4096;
using ftree_type = stxxl::ftree<key_type, data_type, block_size, cache_size>;
```

[5]https://stxxl.org
[6]https://github.com/henrifroese/externa_memory_fractal_tree/blob/master/intro_to_stxxl.md

```
ftree_type f;

f.insert(std::pair<key_type, data_type> (1, 2));
std::pair<data_type, bool> datum_and_found = f.find(1);
std::vector<std::pair<key_type, data_type>> values = f.range_find(0, 1000);
```

3.1 Approach and Design

I approached the implementation mainly by reading and understanding the existing B-Tree implementation in the STXXL. For the insertion algorithm, [5] was of great help.

As a sketch of the design, there is a class `fractal_tree` that implements the Fractal Tree and contains all of the (range-)finding and insertion logic. It holds a mapping from integer IDs to node and leaf objects (for which the tree object is responsible for), and one cache object that is responsible for external-memory interaction. The classes `node` and `leaf` mostly provide getters and setters to work with their data.

3.2 Some `fractal_tree` Class Internals

The `fractal_tree` class is templated with a key type, data type, block size, and cache size. At compile time, the correct node, leaf, and cache types are derived and some static assertions are done to make sure e.g. the cache is big enough to be usable with the tree, etc. When the tree needs a new node or leaf, it creates one on the heap, allocates a new external-memory block for the object, and registers the node/leaf with its mapping from ID to node/leaf. E.g. for nodes:

```
node_type& get_new_node() {
    auto* new_node = new node_type(m_curr_node_id++, bid_type());
    // Register with internal mapping
    m_node_id_to_node.insert(
        std::pair<int, node_type*>(new_node->get_id(), new_node));
    // Allocate new external-memory block
    bm->new_block(m_alloc_strategy, new_node->get_bid());

    return *new_node;
}
```

When using a node/leaf, the tree first `loads` the object, i.e. gets an internal-memory block with the object's data from the cache (which potentially needs to get it from disk) and passes it to the node/leaf. E.g. for leaves:

```
void load(leaf_type& leaf) {
    bid_type& leaf_bid = leaf.get_bid();
    leaf_block_type* cached_leaf_block = m_leaf_cache.load(leaf_bid);
    leaf.set_block(cached_leaf_block);
}
```

When writing to a node/leaf, the tree needs to note that for the cache by doing `m_dirty_bids.insert(node.get_bid())`.

Apart from that, the `fractal_tree` class does not deal with external memory at all, so implementing insertion/finding is possible without needing to think a lot about external memory.

Getting the insertion procedure right for all edge cases turned out to be a little difficult. Usually, insertions just go straight to the root node's buffer. However, if the root node's buffer is full, it needs to be flushed, i.e. the buffer items need to be distributed to the appropriate child nodes' buffers. The child nodes' buffers can again be full, so they need to be flushed, and so forth. When flushing a buffer of a node n just above the leaves, a leaf can overflow and thus need to split, which leads to n gaining a pivot. The node n might in turn overflow and need to split, and so forth. Flushing a few items, then needing to split, continuing flushing etc. back and forth does not lend itself to a nice implementation.

I thus adopted the *small-split invariant* from [5]: Before flushing a node, check if its pivots are at least half full. If yes, split the node. Now when flushing the node, for each child, the node gets at most one additional pivot from each child, so the node will not need to split again.

In comparison, the search procedures were much easier to implement and there were no big roadblocks.

3.3 `node` and `leaf` Internals – Memory Layout and Footprint

A `leaf` or `node` object holds its integer ID assigned by the tree object, a block ID provided by the STXXL to note where the leaf or node's data is stored on disk, and counters for how many buffer items and pivots the node or leaf currently contains. To access their data, the objects hold pointers that can be set to point to an in-memory block holding the object's data, e.g. for nodes:

```
void set_block(block_type* block) {
    m_block = block;
    m_values = &(m_block->begin()->values);
    m_nodeIDs = &(m_block->begin()->nodeIDs);
    m_buffer = &(m_block->begin()->buffer);
}
```

The STXXL provides a block interface that allows a user to read and write from/to external memory blocks holding instances of one type. For example, we can define a block of 4096 bytes that can hold integers and write from/to it like this:

```
using block_type = stxxl::typed_block<4096,int>;
// Get the block manager
stxxl::block_manager* bm = stxxl::block_manager::get_instance();
// Allocate internal memory block
block_type* im_block = new block_type;
// Write to the internal memory block
for (int i=0; i < block_type::size; i++)
    (*im_block)[i] = i;
// Get BID for external memory segment to write to
block_type::bid_type bid = block_type::bid_type {};
bm->new_block(stxxl::default_alloc_strategy(), bid);
```

```
// Write to external memory
im_block->write(bid)->wait();
```

For the node implementation, there were several possibilities to lay out the data in external memory blocks. Each node has pivots, buffer items, and node IDs (identifying the node's children).

The first (and implementation-wise easiest) option was to use a separate block for pivots, one for buffer items, and one for node IDs. This would lend itself to a straightforward usage of the stxxl blocks as above, but would of course increase the I/Os by a factor of 3 immediately.

The second option, which I chose, was to use one block for pivots, buffer items, and node IDs together. This is definitely the most natural implementation that stays closest to the Fractal Tree definition. However, it's not immediately clear how this can be implemented with `stxxl::typed_blocks` as above, as these blocks hold instances of one type.

I chose to use a struct that contains three fixed-size arrays (one each for pivots (which are called `values` in the code), buffer items, node IDs) that is as big as possible while still fitting into a `typed_block` of given block size (i.e. one block holds one big struct):

```
struct node_block {
    std::array<value_type, max_num_buffer_items_in_node> buffer {};
    std::array<value_type, max_num_values_in_node>       values {};
    std::array<int,        max_num_values_in_node+1>     nodeIDs {};
};

using block_type = stxxl::typed_block<RawBlockSize, node_block>;
```

The buffer and values are kept sorted, so using a data structure with logarithmic insertions instead of an array with linear insertions would have been faster as soon as the data is in memory, but I decided to optimize for I/Os and thus not accept the incurred overhead of e.g. using `std::map` for the buffer or pivots.

Having the data nicely accessible in a struct that fills a whole block is great to work with and has very little overhead, but a little difficult to get working:

The struct should fill up one block as much as possible, but of course must not be bigger than the block size. C++ adds padding inside structs to keep the members and the struct itself aligned[7], so the size of the struct is not just the sum of the member sizes, and calculating `max_num_buffer_items_in_node` and `max_num_values_in_node` had to be done at compile-time, this took some time.

I finally opted to separate the calculation of the number of items into a separate class `node_parameters`. The buffer items and pivots are both of `value_type`, and the node IDs are `int`s. First, the class uses a compile-time square root approximation adopted from here[8] to calculate the number of pivots $\approx \sqrt{B/\text{sizeof(value\_type)}}$. Then, a struct is defined that holds the pivots and children IDs, but not yet the buffer:

[7]To preserve portability, I did not want to force packing of the struct with e.g. `#pragma pack` which could lead to unaligned memory accesses that fail on some architectures.
[8]https://gist.github.com/alexshtf/eb5128b3e3e143187794

```
enum {
    max_num_values_in_node =
    static_cast<int>(
            SQRT(static_cast<double>(raw_block_size / sizeof(value_type)))
    )
};
struct node_block_without_buffer {
    std::array<value_type, max_num_values_in_node>       value {};
    std::array<int,        max_num_values_in_node+1>     nodeIDs {};
};
```

Given the block size, the value type and the size of the struct without the buffer, we can now calculate how many buffer items can fit into one block to fill it up as much as possible without overflowing:

```
template<typename value_type,
         unsigned raw_block_size,
         unsigned size_without_buffer>
unsigned constexpr NUM_NODE_BUFFER_ITEMS() {
    unsigned remaining_bytes_for_buffer = raw_block_size - size_without_buffer;
    // The struct without the buffer contains an array of ints (the nodeIDs)
    // and an array of value_type (the pivots) -> it's already aligned to the
    // bigger of the two.
    unsigned alignment = alignof(value_type) > alignof(int)
                       ? alignof(value_type) : alignof(int);
    // Can only fill multiples of alignment
    // -> find biggest multiple of alignment that's <= remaining bytes.
    unsigned max_fillable_bytes =
        remaining_bytes_for_buffer - (remaining_bytes_for_buffer % alignment);

    unsigned max_num_items = max_fillable_bytes / sizeof(value_type);
    return max_num_items;
}
```

Calling this function with the sizes calculated before gives the number of buffer items:

```
max_num_buffer_items_in_node =
    NUM_NODE_BUFFER_ITEMS<value_type,
                          raw_block_size,
                          sizeof(node_block_without_buffer)>()
```

We now know the maximum number of buffer items and pivots and can build the struct with buffer items, pivots, and children IDs. This[9] resource was very helpful to understand struct padding and alignment.

[9]http://www.catb.org/esr/structure-packing/

3.4 Caching

I implemented a standard LRU cache. On initialization, it allocates a pool of a given number of internal memory blocks of given type. Internally, the cache keeps a mapping from block IDs to linked list nodes containing the block ID's current internal memory block.

When a block ID's data is requested from the cache, the cache checks if it's already in internal memory through its mapping. If yes, then it is moved to the beginning of the linked list and its internal memory block is returned. If not, an unused internal memory block loads the data and the block ID and block are inserted at the beginning of the linked list, and the block ID and linked list node are added to the internal mapping. If no unused internal memory block is available, the least recently used item (which is at the end of the linked list, as each time an item is requested, it gets moved to the front) is evicted from the cache.

The cache has no knowledge of nodes/trees/leaves, it works directly with block IDs. It gets a `std::unordered_set` of dirty block IDs that is maintained by the cache's owner (i.e. the tree object) injected in its constructor. When evicting a block ID from the cache, the cache checks if the block ID is dirty (i.e. contained in the set), and only writes the block to external memory if that is the case.

3.5 Some Challenges and Lessons Learned

This was my first time implementing external-memory algorithms and data structures, so I briefly want to mention some aspects I found particularly annoying and/or challenging.

- Debugging and testing: Tracking down and fixing bugs and writing complete unit and integration tests proved more challenging than I was used to, as some branches are only run when the data is already too big to easily grasp in a debugger. My development speed increased when I added a simple `visualize` method that prints a level-order traversal of the tree with all keys and smallest and biggest buffer items.

- Recursive external-memory algorithms: Keeping just some data in-memory per level in the recursion quickly lead to stack overflows for me; I underestimated that and it took some time for me to rewrite parts to keep as little as possible in memory.

- Limited cache: A whole bunch of bugs I spent a lot of time on were due to being at a node n, then calling a method on another node, then wanting to use n again and forgetting to `load` n again as its data was kicked from the cache, for example

```
// Flush child buffer.
if (curr_depth == m_depth - 2)
    flush_bottom_buffer(child);
else
    flush_buffer(child, curr_depth+1);

// Reload (nodes might have been kicked out of memory
// in the recursive call to flush_buffer)
load(child);
load(curr_node);
```

4 Benchmarks

I ran experiments on a Intel i5 8-core 1.6 GHz CPU with x86-64 architecture, using Linux Ubuntu 18.04 with 4 kB block size. I used the STXXL disk configuration option *memory* to keep all data in RAM for faster experiments. Measurements were done using the STXXL I/O performance counter [10].

All experiments were done with a key and data type `int`, cache size of 32kB, and inserted data sizes of powers of 2 from 32kB to 32MB. I performed all experiments on the Fractal Tree, and the B+-Tree (a B-Tree that keeps all data in the leaves, which form a linked list) that was already implemented in the STXXL. On my machine, a key-value pair has size 8 bytes.

I ran the following experiments:

- Sequential Insertion: For a given data size of N bytes, insert the pairs $(1, 1), \ldots, (N/8, N/8)$ into the tree.
- Random Insertion: For a given data size of N bytes, insert a random permutation of the pairs $(1, 1), \ldots, (N/8, N/8)$ into the tree.
- Sequential Search: For a given data size of N bytes, insert the pairs $(1, 1), \ldots, (N/8, N/8)$ into the tree (before starting the measurement). Then sequentially search for keys $1, \ldots, N/8$.
- Random Search: For a given data size of N bytes, insert a random permutation of the pairs $(1, 1), \ldots, (N/8, N/8)$ into the tree (before starting the measurement). Then sequentially search for the keys in the order of the permutation.
- Range-Search: For a given data size of N bytes, insert a random permutation of the pairs $(1, 1), \ldots, (N/8, N/8)$ into the tree (before starting the measurement). Then search for the range $(1, N/8)$.

To compare the number of reported I/Os to the theoretical complexities more fairly, I tried to incorporate the constants intrinsic to the experimental setup by rounding results up/down appropriately and dividing instances of N and B by 8, as only $N/8$ key-value pairs fit into the tree and $B/8$ into a block, because each pair occupies 8 bytes. I thus used the following 'predicted' I/Os:

[10] https://stxxl.org/tags/master/common_io_counter.html

Type	Insert(N)	Search(N)	Range-Search(N)
B+-Tree	$\frac{N}{8} \cdot \left\lceil \log_{B/8}(\frac{N}{B}) \right\rceil$	$\frac{N}{8} \cdot \left\lceil \log_{B/8}(\frac{N}{B}) \right\rceil$	$\left\lceil \log_{B/8}(\frac{N}{B}) \right\rceil + \frac{N}{B}$
Fractal-Tree	$\frac{N}{8} \cdot \frac{1}{0.5 \cdot \left\lfloor \sqrt{\frac{B}{8}} \right\rfloor} \cdot \left\lceil \log_{B/8}(\frac{N}{B}) \right\rceil$	$\frac{N}{8} \cdot \frac{1}{0.5} \left\lceil \log_{B/8}(\frac{N}{B}) \right\rceil$	$\frac{1}{0.5} \cdot \left\lceil \log_{B/8}(\frac{N}{B}) \right\rceil + \frac{N}{B}$

Table 2: Predicted I/Os for the experimental setup (with the factors still visible and not shortened so it can be quickly compared to the complexities from table 1. For the insertion and search predictions, note that when testing e.g. Search(N), we search for all N/8 keys in the tree, giving the factor N/8. For range search, we range-search once for the full range of values in the tree, so the number of items in the result is N/8, and of course (N/8)/(B/8) = N/B.

4.1 Insertion

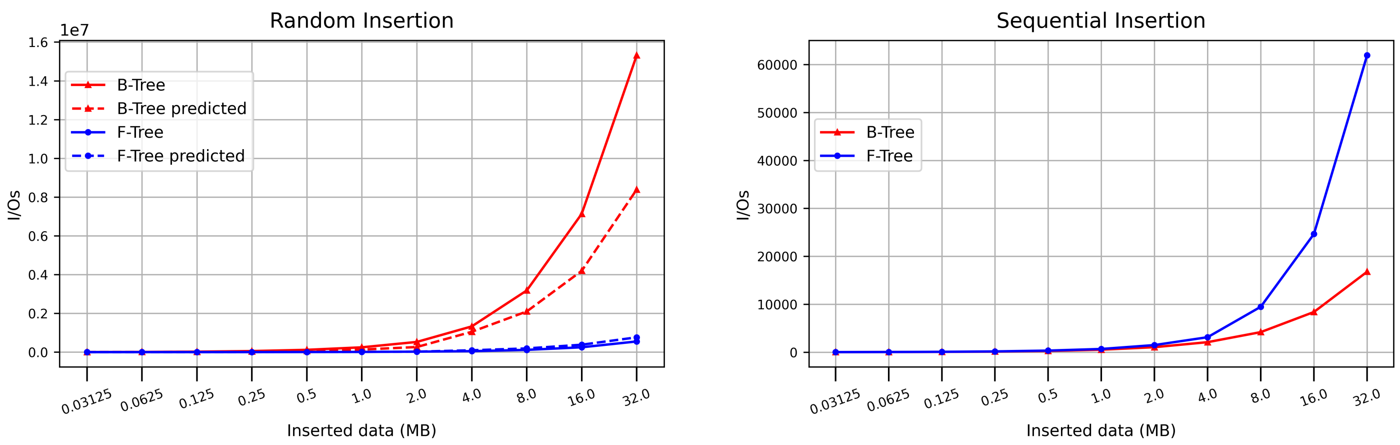

Figure 1: Predicted and measured I/Os for random and sequential insertion for B+-Tree and F-Tree.

Figure 1 shows the experimental results and the predictions. Random insertion matches the predictions very well. The B+-Tree insertion takes on average $\approx$ 29.5 times more I/Os than the Fractal Tree insertion. The predicted factor is $0.5 \cdot \left\lfloor \sqrt{\frac{B}{8}} \right\rfloor = 11$. Due to caching, the Fractal Tree performs slightly better than predicted.

In sequential insertion, both trees perform significantly better than in random insertion due to caching. While the Fractal Tree performs on average $\approx$ 11 times better than with random insertion, the B+-Tree performs on average $\approx$ 536 times better.

That is because as far as I could gather from the B+-Tree implementation, for sequential insertions, new values are simply inserted into the rightmost leaf each time, leading to performances more similar to a linked list with approximately N/B I/Os to insert N elements.

4.2 Search

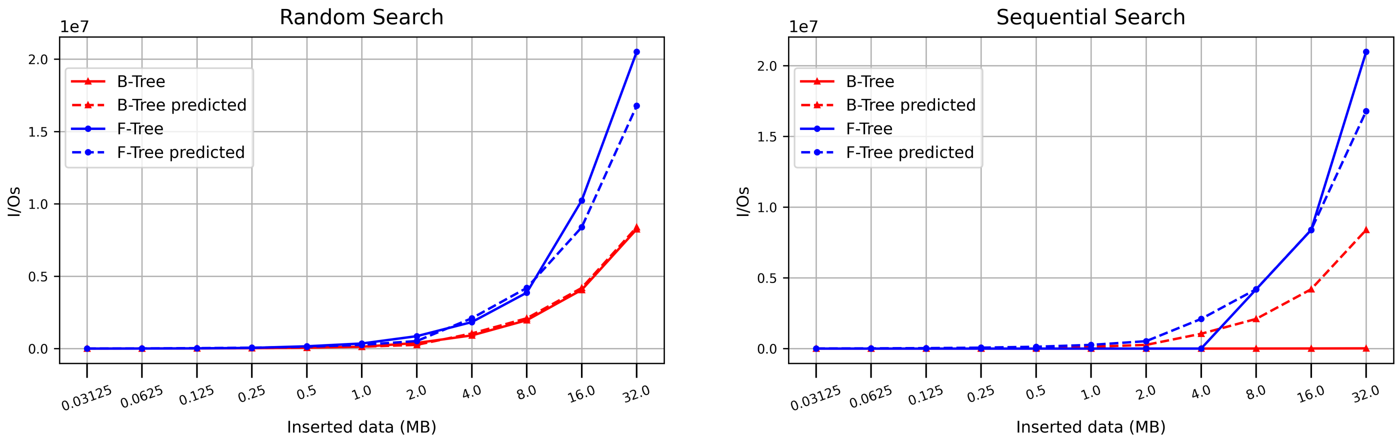

Figure 2: Predicted and measured I/Os for random and sequential search for B+-Tree and F-Tree.

Figure 2 shows the results. Again, random search matches the predictions very well. The F-Tree search takes on average ≈ 1.91 times more I/Os than the B+-Tree insertion, which is close to the predicted factor of $1/\epsilon = 2$.

In sequential search, the fractal tree initially performs much better than predicted, and then deteriorates towards performance similar to random search. A quick manual inspection showed that this is the case as soon as one root-to-leaf traversion of the Fractal Tree does not fit into the cache anymore, because then at least one I/O is necessary for each search, even if most searches go towards the same leaf as the search before.

Similar to sequential insertion, as the B+-Tree keeps all values in the leaves and the leaves form a linked list, the B+-Tree needs only very few I/Os to find the data. Because the leaves are sorted and (at least) the last-visited leaf is always in cache, the B+-Tree can check that leaf first and find the next value. Thus, the N sequential searches can be done in around N/B I/Os.

4.3 Range-Search

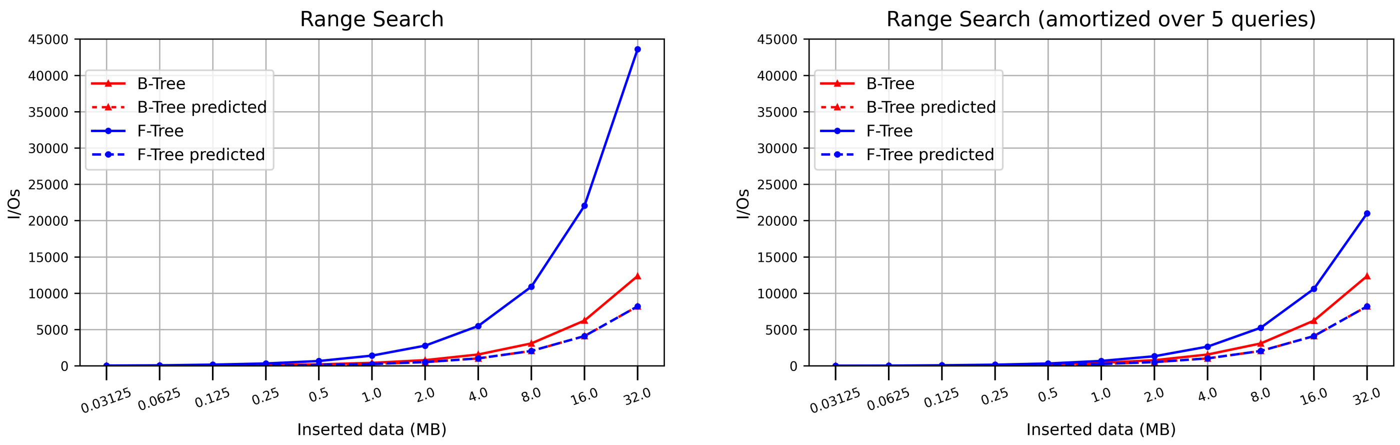

Figure 3: Predicted and measured I/Os for range-search in B+-Tree and F-Tree. On the left, one range search query is performed for each measurement. On the right, the reported I/Os are the average of five consecutive queries.

The left plot of figure 3 shows that for a single range-search query, the B-Tree performs close to the predictions while the Fractal Tree performs significantly worse. This is due to the fact that each call to range-search in my implementation first flushes all buffers along the way, so the first call to range-search is very expensive. Amortized over 5 queries, the Fractal Tree performance is much more in line with the predictions.

References

[1] Alok Aggarwal and S Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] Adam L Buchsbaum, Michael H Goldwasser, Suresh Venkatasubramanian, and Jeffery R Westbrook. On external memory graph traversal. In *SODA*, pages 859–860, 2000.

[3] Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *SODA*, volume 3, pages 546–554. Citeseer, 2003.

[4] Michael A Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C Kuszmaul, Donald E Porter, Jun Yuan, and Yang Zhan. An introduction to b-trees and write-optimization. *Login; Magazine*, 40(5), 2015.

[5] Jelani Jelani Osei Nelson. *External-memory search trees with fast insertions*. PhD thesis, Massachusetts Institute of Technology, 2006.