Report - ALEX: An Updatable Adaptive Learned Index

Data Management Data Structure

https://github.com/owengombas/alex-go

Owen Gombas[1,2]

[1] *University of Bern*
[2] *University of Fribourg*
University of Neuchâtel
Swiss Joint Master of Science in Computer Science

29.05.2024

1 Introduction

Learned indices shift the paradigm by treating indexing as a prediction problem instead of using a predetermined structure to locate keys. It uses models (functions) to learn the distribution of the data and predict key positions. This approach can significantly reduce the lookup time and space complexity, as the model adapts to the specific distribution of the data it manages. The idea is to replace traditional hierarchical or hash-based lookups with a model that can directly predict where to find a key, leading to potentially faster and more space-efficient indices.

However, the main issue with works published before ALEX is the handling of insertions and the time required for bulk loading. For instance, RadixSpline and Recursive Model Index (RMI) [Kraska et al., 2018] use predefined models that can be stacked on top of each other to build a hierarchical model tree, allowing the index to capture more complex data distributions. However, because these nodes are built in a top-down fashion, it becomes very complicated to reflect local changes in the Cumulative Distribution Function (CDF) whenever a new key is inserted. Therefore, any modification to the CDF necessitates rebuilding the entire tree, making these structures immutable.

These data structures demonstrate impressive lookup performance but perform poorly when it comes to insertions. This limitation is precisely what ALEX aims to overcome. By allowing for dynamic updates without necessitating a complete rebuild, ALEX provides a more flexible and efficient solution for learned indices, maintaining high performance even when insertions are frequent.

2 Methodology

Implementing the insertion primitive was essential, otherwise it would have been a simple RMI implementation. Microsoft published their reference implementation to support the original ALEX paper, which was a great help in successfully coding my Go implementation.

While it would have been possible to create a simpler version of a learned index without any reference implementation, ALEX contains an incredible number of corner cases that need careful handling. Missing even one of these corner cases could result in a completely non-functional implementation, as every part of the primitive must work perfectly. Achieving this level of precision and detail using only the paper would be very complicated and time-consuming.

Therefore, the best way to succeed in this task was to refer to the C++ code. This involved thoroughly examining and understanding each

function, porting them to Go, simplifying certain concepts (such as the Bitmap), and selecting the appropriate methods for the operations I focused on (sequential inserts and lookups). This approach ensured a robust and accurate implementation. To obtain a viable final product, I employed the following strategy:

1. Carefully read the ALEX paper multiple times to thoroughly understand its concepts and methodologies.

2. Matched the paper's paragraphs and concepts to the corresponding blocks of code in the reference implementation provided by Microsoft.

3. Understood the behavior and intended goal of each original code block. This understanding enabled me to refactor the code using Go's features. However, even when some code blocks could be simplified, I preferred to maintain a similar structure to the original implementation for two reasons; Firstly to ensure a working implementation within the time constraint; Secondly, to facilitate easier comparison (of my code to the original implementation) and issue reporting for those reviewing my GitHub repository.

4. Ensured that during runtime, using a debugger, the behavior of both implementations matched and that the attributes of the nodes held the same values.

5. Wrote unit tests and benchmarks to verify that the implementation met the expected performance and functionality criteria.

3 Design choices

3.1 Rust first draft

Initially, I intended to implement ALEX in Rust to provide a reliable, memory-efficient, and fully functional product to the open-source community. However, due to time constraints, I decided to stop approximately halfway through and restart from scratch in Go. I used this project as an opportunity to learn Rust, but I quickly realized that the task was more challenging than anticipated.

The difficulty was mainly due to the constraints Rust imposes when implementing a tree-based structure[1]. Rust's borrowing and mutability constraints, as well as the need to rethink the entire internal data structure to perform operations equivalent to C++'s `static_cast`, added complexity. For instance, if I had a Model Node with an array of references to other nodes (either Model Nodes or Data Nodes), casting one of the children to a specific type required significant restructuring using Rust's structs and enums, which would have been very time-consuming.

3.2 Bitmap

The original paper used an array of `uint64_t` to store the bitmap, allowing them to use only 1 bit per cell to indicate the presence of a key in certain position within the key slots gaped array. This avoid wasting wasting 8 bits by using a `bool` type. This make the implementation very computationally and memory efficient, to identify the closest gap for instance we can use bit shifting[2] and use CPU specific instruction to make it even faster and reduce the number of necessary cycles.

However this adds a level of complexity and since we do not aim for a production ready product, I decided to simplify this aspect. Therefore I used an array of `bool` matching the size of the keys array, whenever `bitmap[pos] == true` this means a value is present at `pos` in the keys array. To assess the performance using different Bitmap implementations, I made ALEX agnostic to the Bitmap implementation. This allows anyone to use any kind of Bitmap as long as it meets the interface requirements. Consequently, I could compare my naive Bitmap implementation to a more advanced one provided by an external package called `bitmap`[3], which uses SIMD (vectorized instructions) to enhance efficiency. However, in all benchmarks, I chose to assess the performance with my naive Bitmap because it provides a fairer comparison with my B+Tree implementation, which doesn't use any kind of optimization.

[1] https://www.reddit.com/r/rust/comments/12pljg8/whats_is_a_rusty_way_to_implement_sharable_trees/

[2] https://github.com/microsoft/ALEX/blob/4370da6aa8b509fdc9b0d2c49faa0624b0078589/src/core/alex_nodes.h#L1932

[3] https://pkg.go.dev/github.com/kelindar/bitmap

3.3 Be careful of large keys

The following operation is performed while fitting the linear model:

Given a set of keys:
$K = \{k_1, k_2, k_3, ..., k_n\}$

The following calculation is performed:
$Y = \sum_{k \in K} k^2$

In pseudo-code it translates into:

```
K: int64[] = KEY_ARRAY
Y: float64 = 0
for k in keys:
    Y += k*k
```

It could easily happen that the sum of all keys squared overflow the maximum number Y could hold. That why in the reference implementation the authors used a type `long double`[1] from the C++ standard:

There are three floating point types: float, double, and long double. The type double provides at least as much precision as float, and the type long double provides at least as much precision as double. The set of values of the type float is a subset of the set of values of the type double; the set of values of the type double is a subset of the set of values of the type long double. The value representation of floating-point types is implementation-defined. Integral and floating types are collectively called arithmetic types. Specializations of the standard template std::numeric_limits (18.3) shall specify the maximum and minimum values of each arithmetic type for an implementation..

The implementation of `long double` depends on the compiler and also the architecture of the CPU it runs on, referring to this StackOverflow question https://stackoverflow.com/questions/14221612/difference-between-long-double-and-double-in-c-and-c. But in general a `long double` can hold larger values than a simple `double` type.

However the largest floating point type provided by Go is the `float64` type. So I chose to stick with this type, otherwise it would have been possible to use the type `Float` from the math/big[2] package provided by Go's standard library.

4 Tests

To correctly assess the effectiveness of ALEX and validate its correct behavior, I have implemented several unit tests, that test the sequential insertion of N keys and the retrieval of these keys.

There are four distinct cases, each corresponding to 1k, 10k, 100k, and 1m keys. In these tests, the generated keys are inserted sequentially. After all keys have been inserted, if every went correctly, lookups are performed for each key to verify that the payload matches the expected value. The payload is defined as the size of ALEX (in terms of the number of keys) at the time the key was inserted.

Additionally, using a debugger, I manually verified that the tree structure, node attributes, and the parameters (a and b) of their linear model corresponded to those provided by Microsoft's reference C++ implementation. Although this method is not optimal, it was one of the most reliable ways to ensure a perfect match of my implementation to the reference one given the time constraints. This manual verification process provided confidence that my implementation behaves correctly.

5 Benchmarks

5.1 B+Tree reference implementation

To evaluate the performance of my ALEX implementation, I compared it against a B+Tree implementation I developed, available at my GitHub repository[3]. This comparison is meaningful since both data structures were written in the same language, Go.

The benchmarks measure the time required to sequentially insert 1k, 10k, 100k, and 1m keys without using bulk loading. The keys are randomly generated using the `rand` package from the `math` standard library in Go. For reproducibility, a seed value of 42 was used to generate the random keys.

[1] https://github.com/microsoft/ALEX/blob/4370da6aa8b509fdc9b0d2c49faa0624b0078589/src/core/alex_base.h#L143C3-L146C27

[2] https://pkg.go.dev/math/big

[3] https://github.com/owengombas/datastructures

Both benchmark measure the nanoseconds it requires per oprations (ns/op) for inserting/looking up for a certain amounf of keys.

5.2 B+Tree vs ALEX inserts performances

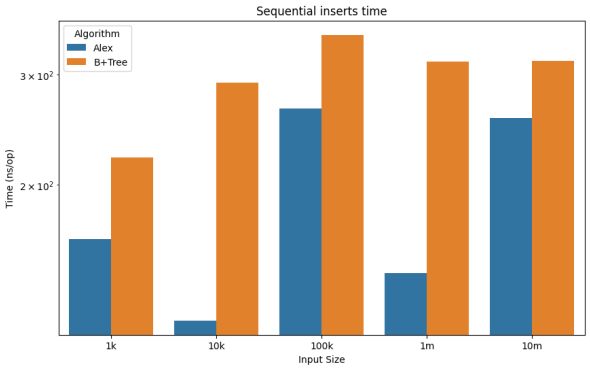

Figure 1: Total time in seconds to perform N sequential inserts in ALEX and B+Tree (log scale).

6 B+Tree vs ALEX lookup performances

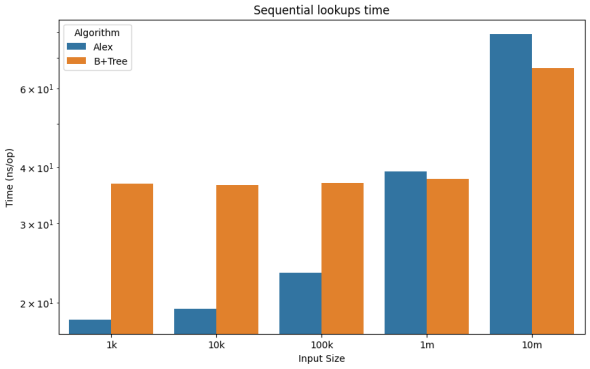

Figure 2: Total time in seconds to perform N sequential lookups in ALEX and B+Tree (log scale).

7 Conclusion

My implementation benchmarks shows that the insert performances are not quite constant while performing sequential insertions; this is due to the fact that the dataset insert is different and therefore the distribution of the keys might helps to improve the time required for inserts (in terms of ns/op).

For lookup we can see that ALEX performs better than a B+Tree but this is not a general rule, ALEX lookup time tends to increase as the number of keys stored in ALEX increase. While B+Tree seems constant until to breaks at 10m keys stored. In the benchmarks, we observe that ALEX performs worse initially but shows less variability as the number of keys increases.

It might not totally reflect the results from the authors but it is likely due to the implementation trade-offs I made to prioritize the educational goals of this project.

In my opinion, ALEX is a great data structure with a great potential. However, it involves numerous corner cases that need careful handling, making it risky to deploy in production environments. If a single corner case isn't correctly handled or if the precision of a specific floating point variable isn't sufficient, the entire structure can fail. Additionally, type overflow can occur easily. Despite these challenges, ALEX promises excellent performance and helps us imagine a bright future for these type of data structures.

References

[Ding et al., 2020] Ding, J., Minhas, U. F., Yu, J., Wang, C., Do, J., Li, Y., Zhang, H., Chandramouli, B., Gehrke, J., Kossmann, D., Lomet, D., and Kraska, T. (2020). Alex: An updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD/PODS '20. ACM.

[Kraska et al., 2018] Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. (2018). The case for learned index structures.

Appendix A