README

Hello everyone. Today I'll be presenting our group project on replicating the research paper 'ALEX: An Updatable Adaptive Learned Index' from SIGMOD 2020. We are Group 31. In this video, I'll explain why we chose this paper, our implementation approach, and the key results we obtained.

First, let me explain why learned indexes matter. Traditional database indexes like B+Trees have served us well for decades, but they have limitations in modern memory-intensive applications. They require multiple comparisons and pointer jumps, which become bottlenecks in pure in-memory scenarios.

In 2018, researchers introduced the concept of learned indexes, which use machine learning models to predict data locations. However, these early versions couldn't handle updates efficiently - they were essentially read-only structures.

That's where ALEX comes in. Published in 2020, ALEX represents a breakthrough - it's an adaptive learned index that supports updates while maintaining high performance.

For our project, we implemented a simplified version of ALEX in Python, along with two comparison baselines: a binary search index and Python's built-in dictionary.

We made several design choices to keep our implementation manageable within the project scope. We maintained the core 'tree of linear models' structure but simplified the adaptive mechanisms. Our version uses fixed tree depth and basic split policies, focusing on single-threaded in-memory operations.

Let me briefly show our code structure. Here you can see our three main classes: SimpleALEX for our learned index, BinarySearchIndex, and we use Python's dict as our third baseline."

Now let's look at our experimental results. We tested with three dataset sizes: 1,000, 5,000, and 10,000 records.

Figure 1 :First, construction time. ALEX takes longer to build than the other indexes, especially as data size increases. This is expected since it needs to train models.

Figure 2:For memory usage, ALEX and binary search are quite similar, while Python dictionaries use more memory due to their hash table implementation.

Figure 3:For point queries, Python dictionaries are fastest thanks to hash-based O(1) access. However, as data grows, ALEX closes the gap with binary search, showing how learned models become more effective with larger datasets.

Figure 4:This is where ALEX really shines. For range queries, ALEX significantly outperforms both binary search and dictionaries, especially at larger scales.

This validates the paper's claim that learned indexes excel at workloads involving ordered data access.

Our results confirm the paper's main findings: ALEX offers superior range query performance while supporting updates, though it comes with higher construction costs.

We faced several challenges during this project. The original paper used optimized C++ code, while we worked in Python, so absolute performance numbers aren't comparable. However, the performance trends align well with the original research.

This project taught us how machine learning can enhance traditional systems, and how research ideas translate into practical implementations.

In conclusion, we successfully replicated the core concepts of ALEX and verified its performance advantages for range queries. Learned indexes represent an exciting frontier where machine learning meets database systems, potentially revolutionizing how we store and access data.