# PMAlex

A PMA backed updatable Adaptive Learned indEX

Parker Beckett, Matt Myers, James Rider
Github: https://github.com/Tweal/PMAlex

## Introduction:

ALEX is an updatable learned index implementation released in mid 2020. It is a tree structure that uses simple linear models for predicting key locations and insertion costs. It uses the cost models to effectively manage splitting and tree structure manipulation while leveraging the model predictions to ensure efficient inserts and retrievals. It was designed using a gapped array backend with model based inserts determining where the gaps are placed. This provides a O(log(N)) lookup time, an expected insert time of O(log(N)), and a worst case insert of time O(N). They attempt to minimize this worst case scenario by tracking statistics of the performance and splitting the nodes if it deviates too far from the expected performance. However, if there is a non-static distribution the performance will degrade to linear time as the model fails to predict location well. Our project was to swap out this gapped array backend for a packed memory array (PMA). A PMA is also based on including gaps within an array but rather than using the model to place the gaps they are evenly distributed across leaf nodes of the implicit tree structure. As objects are inserted, the PMA will redistribute the gaps within adjacent leaf nodes as certain density bounds are exceeded. Additionally it will double the size of the array if it starts to run out of gaps. These redistributions ensure that there is almost always a nearby gap when inserting elements and guarantees a worst case insert time of O($\log^2$(N)). The lookup and expected insert times remain O(log(N)).

Our initial proposal involved implementing a simple learned index with a PMA backend from scratch but was later updated to utilize provided code from the original ALEX paper and a graph based PMA implementation. After gaining an understanding of both code bases we thought of abstracting out the critical infrastructure of the AlexDataNode class in order to leverage inheritance to write a wrapper for the PMA code. Doing so would allow us to easily swap the backend of ALEX to observe the differences in performance of different types of storage. We were also interested in exploring situations in which the PMA implementation should outperform the existing implementation such as non-static workloads. Such scenarios should force both implementations to degrade to their worst case inserts which would result in the PMA version performing significantly better.

The authors of the ALEX paper mention that they attempted a similar implementation with a PMA backend but found that the use of such a backend negated the performance benefits of the model based inserts as the redistributions move the keys away from their predicted position. As such we expected the performance of our implementation to be worse than the existing ALEX code in standard workloads (ie static key distributions). However, the authors fail to mention what version of PMA they used so there was a chance that the provided implementation would be more efficient. In addition to a potentially more optimized PMA, the implementation provided makes extensive use of parallelization which could affect performance.

## Benchmarks:

We used the provided benchmarking code provided from ALEX which takes a binary file of keys and generates random payloads. We chose to use the YCSB dataset which provides 200M keys of 8 byte unsigned ints to ensure compatibility with the uint keys of PMA. In order to run the PMA benchmark, we modified main.cpp in ALEX to use a PMA rather than an ALEX index (see the pma-benchmark branch in our repository).
Our benchmarks for ALEX were ran with the following parameters:

- Total number of keys: 200M
- Initial number of keys: 10M
- Batch size: 1M
- Percentage inserts: 10, 30, 50, 70, 90
- Lookup distribution: Zipf

The following results were used as a baseline for our future comparisons:

| Percentage Inserts | Lookups / Second | Inserts / Second | Operations / Second |
|---|---|---|---|
| 10% | 2.423e07 | 5.096e06 | 1.762e07 |
| 30% | 2.513e07 | 5.215e06 | 1.177e07 |
| 50% | 2.551e07 | 5.072e06 | 8.461e06 |
| 70% | 2.499e07 | 4.968e06 | 6.540e06 |
| 90% | 2.443e07 | 5.035e06 | 5.470e06 |

Additionally we established a baseline for the provided PMA implementation with the following parameters and results:

- Total number of keys: 200M
- Initial number of keys: 10M
- Batch size: 1M
- Percentage inserts: 10, 30, 50, 70, 90
- Lookup distribution: Zipf

| Percentage Inserts | Lookups / Second | Inserts / Second | Operations / Second |
|---|---|---|---|
| 10% | 3.870e07 | 5.450e05 | 4.837e06 |
| 30% | 3.833e07 | 4.577e05 | 1.484e06 |
| 50% | 3.864e07 | 4.756e05 | 9.396e05 |
| 70% | 3.872e07 | 4.828e05 | 6.860e05 |
| 90% | 3.872e07 | 4.826e05 | 5.355e05 |

Here we see that the PMA's inserts/second and operations/second are lower by about a factor of 10. Its lookups/second, however, are consistently higher by a factor of 10; this may have to do with ALEX having more overhead than PMA. We also see that ALEX's insert rate appears slightly more stable than that of PMA across different workloads.

## Results:

Unfortunately, we failed to properly gauge the difficulty of modifying the existing alex code with a new backend in the month we allotted ourselves. There were several scheduling conflicts and time balancing issues caused by the employment of team members. We additionally encountered several difficulties that we will outline below that ultimately prevented us from successfully implementing the PMA backend.

The initial difficulty we encountered came from the PMA implementation we were provided. It was originally written for a graph implementation meaning that it had source and destination to act as keys. This proved to not be too much of an issue as we had expected as we found that we could just use the same source for everything and then use the destinations as our keys. The PMA implementation was also very difficult to read and debug as it was written to be highly efficient and highly concurrent. The biggest issues we anticipated was the amount of casting that was required to go from the datatypes used by alex to the uint32_t that is favored by the PMA implementation. In an ideal world we would have been able to use the PMA code as a black box simply interacting with it without needing to modify the code much if at all. However, while writing the wrapper we found that a more in depth understanding of the PMA code proved to be needed as we moved into debugging the issues of our implementation.

Additional modifications were needed to facilitate the different node splitting operations alex requires. The code as written does not support splits at all and simply expands or contracts as certain density bounds are passed. In order to support splits we needed to add additional functionality to support cost tracking and functions to remove or add ranges of keys. The new add and remove functions did not prove to be too difficult to implement using the existing batch functions. Utilizing the existing iterator for PMA and writing our own DataNode iterator to progress it made range operations not impossible. We had some issues with the cost functions and stat tracking needed to facilitate them. As we needed to store the stats in the DataNode object but modify them from within the PMA object. Most of them we were able to manipulate in our wrapper code but other calculations we could not as they depend on what the PMA did internally. Additionally we were uncertain what our new cost functions should even look like for the PMA implementation. For example, the empirical cost for the gapped array is a linear combination of the search iterations and number of shifts the inserts took. We can use that first term with PMA but how do we track the shifts or redistribution costs? They should be pretty low, would we be safe to just ignore them entirely? This is the kind of thing that we really should have come and asked in office hours for but we figured these scaling factors for the two terms was something that we would be able to fine tune as we tested to find the optimal values once we had it running.

By far the largest hurdle we needed to overcome was how tightly coupled the alex code was. Indeed, this ultimately proved to be our undoing. In our preliminary probing of the code we found that alex defines an abstract class for AlexNode and then subclasses that with the

AlexModelNode and AlexDataNode. This is what led us to believe that we would be able to extract the critical code for the DataNode and inherit from that in a child class that would act as a wrapper for new backend implementations. Our initial forays into the adaptation looked promising as much of the functionality for handling the data seemed to be easily handed off to PMA. The real problems came once we thought we had finished the new AlexDataNode and moved onto debugging and attempting to run alex. It was at this time that we discovered that the data node was not as nicely encapsulated as we had expected. The variables within were being directly accessed by the other two alex files, not even simply referenced, they were being modified by seemingly unrelated methods. We spent most of the last week trying to untangle these methods before just ultimately giving up and accepting that we were not going to finish it.

At some point near the end we came to the conclusion that we might have been better off scrapping large portions of either the PMA code or the ALEX code and starting nearly from scratch working around their limitations. Parker chose to pursue this option on the last day to see if it would have been viable and will update this section if he gets it working before we need to turn this in.

We likely started later than we should have, starting the coding in earnest around the midpoint, but we don’t feel our failure to complete came down entirely to an issue of time. We made a hard push to try to complete it in the last week, pulling multiple all nighters to try to reach the home stretch. At some point we just got too burned out on the code and gave up. It almost felt futile as we realized that all of our work would have ultimately been wasted and that our best path forward may have been to restart.

We also came to the conclusion that we should have utilized office hours significantly more than the two times that we did. There are a lot of issues that we should have looked for guidance on rather than attempting to solve ourselves. Honestly, our approach in the first place should really have been cleared more explicitly with the instructor. The suggestion that was made after our presentation about using a single PMA to handle the full backend rather than an individual one for each data node sparked a conversation in the team on how much better that solution would have been and how much more interesting that approach could have been to implement. We tried using individual PMAs for each leaf because that is how they did it with the gapped array and we were trying to just provide a one to one replacement.

## Member Contributions:

Parker:
- Alternate PMA implementation exploration
- Assisting compiling and benchmarking
- Debugging

Matt:
- PMA Cost Functions
- PMA Model Statistic tracking
- PMA Benchmarking
- Debugging

James:
- PMA Data Node wrapper
    - Constructors
    - Bulk Load Functions
    - Iterators
    - PMA range functions
    - Insert
    - Remove
    - Model updates and predictions
- Tree level iterators
- Debugging
- Alex Benchmarks