Revisiting Learned Indexes: Polynomial Models for DuckDB

Alfredo Reina Corona
University of Southern California
reinacor@usc.edu

Sanchit Sanjay Kripalani
University of Southern California
skripala@usc.edu

Adnan Mazharuddin Shaikh
University of Southern California
adnanmaz@usc.edu

ABSTRACT

DuckDB is a popular open-source database system designed for efficient Online Analytical Processing (OLAP) workloads, and offers fast query performance in serverless environments. However, its current indexing mechanisms, zone maps and Adaptive Radix Trees (ART) have notable limitations. Zone maps struggle with skewed data distributions, resulting in redundant data scans, while ARTs has notable memory and space overhead. In this project we propose extending DuckDB by integrating polynomial learned indexes, which use polynomial regression models to predict key positions more accurately and compactly than previously used linear model, and traditional structures. By modeling the underlying data distribution, our learned indexes are more adaptive and require fewer pointers, reducing memory footprint. Our solution involves modifying DuckDB's indexing infrastructure by incorporating a pre-built indexing structure called ALEX, and modifying ALEX to support polynomial regression, instead of the traditional linear regression used. We show that ALEX with polynomial regression is more efficient than the base ALEX model up to a certain threshold and has potential to outperform the base version at larger scales.

1 INTRODUCTION

Indexes are special data structures that help in faster and efficient lookup of data without the need to access every row in the database. Therefore, indexes play a key role in improving the performance of a database. Traditional index structures like B+ trees, BW trees, zone maps, Adaptive Radix Trees (ARTs) have been developed over the years to optimize the index performance. B+ trees are row-store databases and use a balanced tree to maintain an ordered set of keys. All values in a B+ tree are stored at the leaf level, while the internal nodes are used as filters to search using key ranges. Zone Maps [14] are a common kind of index found in columnar data stores such as DuckDB. The data is sorted and divided into blocks, with the minimum and maximum values of each block being stored to enable fast filtering during scans. Adaptive Radix Trees (ARTs) are the state-of-the-art indexes used by many database systems including LeanStore [9], HyPer [10] and DuckDB [12]. ARTs improve on the traditional tries by using variables sized nodes and path compression to reduce tree size. While effective in many use cases, all of these conventional indexes suffer from some fundamental limitations.

The major drawback of traditional indexes is the space and memory they consume. Tree based structures like B+ trees and ARTs rely on a hierarchy of nodes to maintain a balanced order. ARTs also use pointer rich representations for adaptive node sizes to reduce the space taken by constant space nodes. However, this results in higher memory consumption per key due to the variable size nodes.

Another issue is poor adaptability to different data distributions. B+ trees and ARTs do not consider data distribution and statistics to optimize the structure. Similarly, Zone maps fail to eliminate redundant scans when data is skewed and clustered [14]. As a result, conventional indexes perform poorly for skewed and non-uniform workloads.

To address these challenges, recent research work [3], [8] explores new type of indexes called learned indexes. These indexes use Machine Learning heuristics to predict the position of searched key in the index. Unlike previous methods, where internal nodes explicitly store keys and pointers to child nodes or data records, learned indexes store key-based statistics in the form of mathematical functions to predict the position of the key. By leveraging the statistical patterns of data, these indexes aim to achieve higher accuracy while maintaining lower memory usage. The Case for Learned Index Structures [8] is one such breakthrough research in the field of learned indexes. The study introduces the concept of Recursive Model Index (RMI), which improves on the traditional tree-based approaches. RMI uses a combination of Neural Networks and Linear Regression model for predicting key location. Although RMIs are able to drastically improve index performance, they do not support inserting and updating the index.

ALEX (Adaptive Learned Index) [3] introduces hybrid structures to support dynamic workloads. Although ALEX was mainly created to improve OLTP workloads, it is observed that the optimizations of ALEX improve the performance in read-only, OLAP workloads. Therefore, ALEX is the state-of-the-art learned index model and hence the basis of this study. One of the major drawbacks of ALEX is that a simple linear regression [11] used in ALEX is not able to capture the complex data distributions of different workloads. This is the exact problem this study aims to solve: to extent the capabilities of ALEX to polynomial regression to improve index performance for different and complex distributions.

We show the strength of Poly-ALEX, in DuckDB, a lightweight, in-process analytical database. DuckDB was designed to keep efficiency and speed in mind. It uses a combination of ART and Zone Maps in the form of a MinMax index. As described previously, these conventional indexes incur excess space and memory overhead. Such an overhead is costly, especially in a lightweight system and therefore can be upgraded on.

Our contributions are threefold: (1) we modify ALEX to support polynomial regression in its model nodes; (2) we integrate Poly-ALEX into DuckDB as an extension, allowing users to choose which index to use; and (3) we evaluate index performance against traditional linear model ALEX and DuckDB's existing ART+MinMax indexes. Our findings show that Poly-ALEX improves lookup throughput by up to 1.5x compared to Linear ALEX and outperforms ART on larger datasets, while maintaining a similar memory footprint. Additionally, index creation time remains low — approximately 1.1x that of Linear ALEX and significantly faster than ART, which takes up to 2x longer to build.

2 CORE STRUCTURAL ELEMENTS

To implement learned indexing within DuckDB, several components of the system must be extended or newly developed. These components work together to integrate machine learning models into the indexing pipeline while maintaining compatibility with DuckDB's existing architecture. In the following sections, we describe the main components used within the system. Their modification and development will be described in Section 3 of this report.

2.1 DuckDB

DuckDB is the result of a one-of-a-kind research to create an embeddable relational database management system tailored for analytical (OLAP) workloads [12]. Unlike traditional monolithic database servers that run as standalone processes and require complex setups, DuckDB [4] is designed to operate entirely within the host application's process like a linked library. This design is therefore well-suited for local computing and smaller edge computing applications like IoT.

DuckDB is heavily inspired by SQLite's performance in transactional (OLTP) workloads while being embedded *in-process*. DuckDB aims to bring a similar level of portability and simplicity to analytical processing (OLAP) - something which SQLite is not optimized for due to its row-major layout and a lack of vectorized execution [7]. Data science workflows use tools like Python's Pandas or R's dplyr for data manipulation. While these tools are easy to use and are widely adopted, they rely on large runtime environment and lack full relational query optimization and efficient execution for large datasets. DuckDB on the other hand, provides SQL-like capabilities with efficient vectorized execution and columnar storage to improve performance on large analytical queries. This coupled with a small footprint and a zero-dependency design make it a powerful tool for embedded analytical data management systems.

However, as discussed in the previous sections, DuckDB uses conventional indexes which have a large memory overhead. Therefore, a learned index, and in particular, a modified version of ALEX [3] called the Poly-ALEX has been developed for this study.

2.2 Learned Indexes

The paper The Case for Learned Index Structures [8] introduces a new technique for database indexing. The authors argue that traditional index structures—such as B-Trees and Hash-maps be viewed as models that map input keys to record locations or existence predictions. They theorize that machine learning models trained to capture the distribution of the underlying data, can replace traditional structures. Learned indexes theoretically and practically show significant improvements in memory usage, lookup speed, and hardware utilization, especially for OLAP workloads.

One of the main contributions of the paper is the design of the Recursive Model Index (RMI). The RMI organizes the models into a tree-like hierarchy, where each level narrows down the search space. The top-level model provides a rough estimation of a key's location within the dataset, have a smaller search space. Simple models, such as linear regressions handle fine-grained predictions at the bottom levels. Each model is trained independently, and focuses on minimizing prediction error within its range. This structure allows the system to combine the benefits of modeling the global data distribution with highly efficient, local fine-tuning.

In their experiments, small feed-forward neural networks where used for the higher levels of the RMI, while linear regression models where used for lower levels. This structure reduces resource usage, and saves space while improving lookup performance as compared to traditional indexing structures.

Learned indexes have notable limitations. They work best with static, read-heavy workloads and struggle under write-heavy workloads due to high retraining and model re-balancing costs. Additionally, changes in the underlying data distribution can degrade model accuracy over time. Among other issues, predictions may not be accurate in range queries; techniques such as biased binary or quaternary search, are often needed to correct for prediction errors.

2.3 ALEX: Updatable Adaptive Learned Index

The ALEX index [3] addresses many limitations of its predecessor. Recursive Model Index (RMI) demonstrated improvements over B-Trees in static, read-only workloads, but lacked support for dynamic updates, making them unusable for most real-world applications involving inserts, deletes, and updates. ALEX builds on the foundational idea of treating indexes as learned models, but introduces several updates to make learned indexes practical for mixed workloads.

ALEX combines the predictiveness of learned models with traditional storage techniques to provide efficient support for point lookups, range queries, inserts, and updates. Its architecture is tree-like, similar to B-Trees, but replaces comparator-based branching with model-based routing. Internally, ALEX uses a variant of RMI, where internal nodes contain linear models that predict which child node to follow, and leaf nodes, called data nodes, contain both the data and a lightweight linear regression model to predict key positions. The models used in ALEX are exclusively linear regressions due to their computational efficiency and small memory footprint, but as explained in section 3, can still lead to sub-optimal performance.

ALEX also introduces adaptive mechanisms to keep models efficient in shifting data distributions. It uses linear cost models to monitor the deviation between expected and actual performance of models during runtime. When deviation becomes significant, the index adapts by either expanding data nodes, retraining models, or splitting nodes—both sideways (as in B-Trees) or downwards (transforming a data node into an internal node with new children). These actions allow ALEX to effectively handle skewed or shifting data.

Furthermore, ALEX supports automatic bulk loading through a greedy top-down RMI construction guided by cost models. This flexibility enables it to scale to large datasets while maintaining low prediction error.

ALEX noticeably improves upon its predecessor, but still has a few notable weaknesses. Although it is resistant to distribution shifts and has usable write performance, insertions into full data nodes may require expensive expansions or splits, especially under skewed workloads. Additionally, the exclusive use of linear models, while efficient, may limit predictive accuracy in non-linear regions

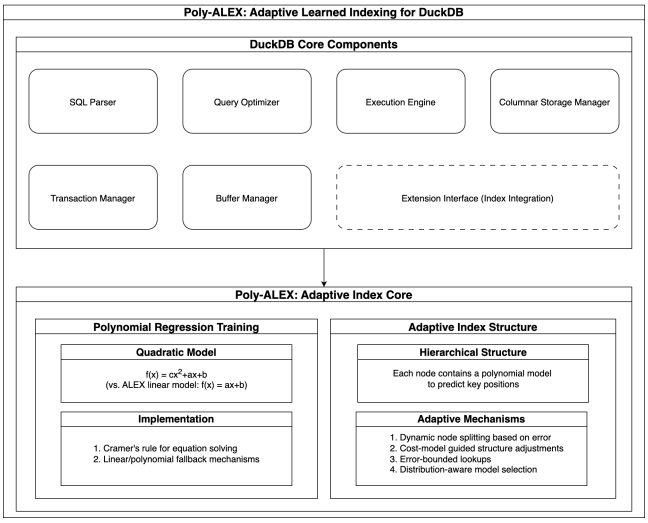

Figure 1: Poly-ALEX integration into DuckDB

of the data or with non-linear datasets in general, even with the adaptive partitioning helping mitigate the issue.

3 IMPLEMENTATION

ALEX was heavily modified to enable support for polynomial regression. DuckDB was modified by adding an extension to the database, allowing Poly-ALEX to function in place of the native indexing models. *Figure 1* demonstrates the system design of our proposed system. DuckDB Extensions have been used to connect to the modified ALEX, that is Poly-ALEX. Poly-ALEX extends the capabilities of ALEX with Quadratic model, and the corresponding Adaptive Mechanisms to support these changes.

3.1 DuckDB Extension

DuckDB extensions [6] are modular components that expand the core functionalities of the database while keeping the system lightweight. These extensions can be loaded at runtime and provide support for a range of features such as indexing, specialized functions such as statistics and integration with other tools and languages. Extensions simplify integration with existing systems and reduces the need for data conversion while interacting with the database, thus making development faster.

The DuckDB extension developed through this research, adds support for integrating learned index structures by introducing a set a custom *PRAGMA* functions that manage index creation, lookup and benchmarking. These functions allow users to create and manipulate learned indexes like Poly-ALEX and traditional structures like ART directly within DuckDB. The extension also provides functionality to benchmark both index creation time and batch lookup performance.

Additionally, the extension provides functionality for loading synthetic benchmark datasets into DuckDB and performing lookup performance evaluations using configurable parameters. This allows users to quantitatively evaluate the efficiency of different index structures under various data distributions and workload patterns. Through these capabilities, the extension not only provides a way to integrate Poly-ALEX index into DuckDB, but also allows the user to run benchmarks and switch between different indexes at runtime.

3.2 Model Class and Builder

3.2.1 Equations. ALEX [11] used a linear model as shown below. By only having to store 'a' and 'b', ALEX is able to maintain a low overhead for these models, enabling fast look-ups and a small memory footprint.

$$f(x) = ax + b \tag{1}$$

Poly-Alex uses a polynomial model in order to provide better predictions for non-linear data, such as data with a gaussian or Zipfian distribution. Specifically, we implement a Quadratic model as it effectively models a large number of commonly used non-linear datasets.

$$f(x) = cx^2 + ax + b = 0 \tag{2}$$

By making 'c' the squared term, we are able to decrease the amount of code to be modified. Although the memory overhead of storing an extra variable is more expensive than the linear model, the flexibility and consistently better performance, make the overhead acceptable.

3.2.2 Polynomial Implementation. While modifying the linear model class to support a polynomial model required only structural changes, edits to the model building class involved several significant enhancements:

- **Higher-order terms** such as $\sum x^3$, $\sum x^4$, and $\sum x^2 y$ were added to compute the parameters of a second-degree polynomial.
- A **3×3 system of equations** was constructed using the *normal equations for quadratic regression*, as a quadratic model requires solving for three coefficients: c (for x^2), a (for x), and b (the intercept).
- **Cramer's Rule** [13] was implemented to solve the system of equations by computing the determinants of the coefficient matrix and its variants. While computationally inhibitive at large scales, it works efficiently for smaller datasets
- A **linear model fallback** is used when there are too few data points to fit a quadratic curve (fewer than 3).
- A **polynomial fallback strategy** is also included: if the determinant of the coefficient matrix is zero (the system is not solvable), the model defaults to $a = 0$, $c = 0$, and sets b as the average of y values.

3.3 Bulk Loading

Bulk loading is the process of inserting large amount of data into a database with a single operation. The original functions where designed for linear models and required heavy modifications to support polynomial models. These changes were essential to allow the index to handle non-linear key distributions efficiently. Polynomial models offer clear advantages in terms of accuracy, their integration introduced additional challenges that needed to be carefully addressed to preserve efficiency of the bulk loading process.

3.3.1 Model Transition. The first major change was the replacement of `LinearModel` with `PolynomialModel` throughout the bulk loading functions. This enabled higher-order fitting, which better captured sub-linear and non-linear patterns that traditional linear models where unable to represent accurately. Using polynomial models during bulk loading, noticably minimized search costs. This transition also resulted in higher computational and memory overhead due to the additional coefficients and more complex evaluation process [13].

3.3.2 Coefficients Initialization. Explicit initialization of the polynomial coefficient `c` was introduced to ensure correct behavior when linear approximations where still superior. In scenarios where linear behavior was optimal or required, such as root or intermediate nodes, `c` was set to zero. This mitigated issues caused by uninitialized higher-order terms, which had the potential to introduce adversary non-linear behavior. By explicitly controlling the polynomial degree, correctness and stability are preserved throughout bulk loading.

3.3.3 Model Propagation. Another modification involved propagating the model during recursive bulk loading and fanout partitioning. In base ALEX, only `a` and `b` were scaled and passed to child nodes. With Poly-ALEX, propagating `c` was required to ensure that the child models accurately reflected the behavior of their parent during bulk loading. This change preserved model continuity across hierarchical levels and ensured that the learned structure remained precise and well-aligned with the underlying key distribution.

3.3.4 Model Reuse in Leaf Construction. Finally, the process of splitting and reusing models during leaf node construction within bulk loading was updated to handle polynomial models. When existing models where reused, all polynomial coefficients are reconstructed and adjusted to fit the new key ranges. This avoids the need to retrain models from scratch during bulk load operations and ensures consistency across splits and merges. This added minor complexity to the code and had additional memory costs, but it significantly improved efficiency and maintained the quality of the learned models.

3.4 Cost Sampling

Cost sampling is a technique used to estimate the expected search and shift costs of a data node by analyzing a subset of keys rather than the entire dataset. It progressively increases the sample size until the measured statistics stabilize. This allows for an accurate approximation of the node's behavior while significantly reducing computational overhead and runtime as compared to a global evaluation.

Significant modifications where made to the cost computation pipeline as well. These changes ensured that cost estimation methods remained accurate and where compatible with the polynomial model. The following sections describe the changes.

3.4.1 Model Transition. Similarly to the bulk loading functions the `LinearModel` was replaced with the `PolynomialModel`. Without this change, the cost estimation logic the additional degree introduced would not have been taken into consideration by the model , which would have led to inaccurate estimations of expected search iterations and shifts. As described in the previous section, the changes add slightly more computational overhead but where necessary to ensure consistent functionality.

3.4.2 Coefficient Initialization. Wherever models were reconstructed or reused for sampling initialization or cost estimation from existing nodes, the polynomial coefficient `c` was explicitly initialized. In scenarios where only `a` and `b` were set, leaving `c` undefined introduced unpredictable behavior during cost prediction. By explicitly copying or zero-initializing `c`, the implementation gives stable and accurate predictions. This makes Poly-ALEX robust and prevents silent estimation errors.

3.4.3 Progressive Sampling. The progressive sampling process incrementally increases sample size to stabilize expected search iteration and shift statistics. This process was also updated in Poly-ALEX. Specifically, during each sampling round, a polynomial model is fitted, or expanded, to match the sampled capacity, making the predictions polynomial compatible. This change makes sure that sampling reflects the true model's prediction behavior and improves the accuracy of the full-size cost estimates.

3.4.4 Prediction and Accumulation in Sampling Helpers. Helper functions used in sampling-based estimation were modified to use `PolynomialModel` for position prediction. These functions simulate data node construction and gather statistics by predicting positions and accumulating the number of expected search iterations and shifts. Switching to polynomial prediction better reflects the behavior of the model. This prevents underestimation or misrepresentation of potential clustering or dispersion in the key distribution, leading to more reliable and realistic cost statistics.

3.5 Other changes

Many of the other changes have similar reasoning to them as the previous sections so we will briefly go over them.

3.5.1 Root Expansion. The `expand_root` function dynamically increases the key domain of the index when a new key falls outside of the current bounds, either to the left or the right. It does this by either expanding the number of children in the current root model node or by creating a new root node that wraps the existing root. The main change required was the explicit handling of the `c` coefficient when creating new model nodes during expansion. Before, only the `a` and `b` coefficients needed to be scaled when creating a new root model, but now the additional `c` term must also be scaled appropriately to maintain the accuracy and continuity of predictions across levels. An uninitialized or incorrectly scaled `c` could lead to incorrect predictions and disrupt the index, especially during lookups and insertions near the domain boundaries. Just as important, when shifting these range (e.g., updating `b` when expanding to the left), assumptions that `c` should remain zero in certain cases were explicitly retained to avoid introducing unintended non-linear behavior, similarly to bulk loading. These modifications ensure that expansion remains compatible with the polynomial models. With these changes the tree can accurately and safely expand while making polynomial-based predictions.

3.5.2 Downward Splitting with Polynomial Model Support. The `split_downwards` function replaces a full data node with a model

node and splits the key space beneath it. It follows the structure outlined by the fanout tree. This allows the index to dynamically grow and organize the data hierarchically as new keys are inserted. In base ALEX, only linear models were used, so the newly created model node used linear calculations for determining its parameters `a` and `b`. This logic was modified to handle quadratic parent nodes. The updated implementation introduces an inversion operation to invert the parent to determine the key range boundaries corresponding to the split buckets. This lets the new child model maps its key range back to global positions. If the parent model is linear or constant (when `c` is zero), the inversion reduces to simple algebra. When `c` is non-zero, the quadratic equation is solved directly, ensuring correctness even in the non-linear case. Notably the new child model still defaults to a linear form (`c = 0`) to avoid unnecessary complexity in the subtree unless higher-order modeling is required later. These changes maintain correctness during splits under polynomial parent models, and also preserve efficiency in the cases where linear splitting suffices, which happens to be the majority of cases.

3.5.3 Data Node Splitting. The `create_two_new_data_nodes` function splits an existing data node into two new data nodes and inserts them as children of a parent model node. This happens when the index structure must divide the key space more finely, often after a downward split. In the original version, determining the split point was done by directly mapping the midpoint bucket ID to a key value using the inverse of a linear model. With polynomial models this had to be changed. The updated version implements a polynomial inversion step to compute the split key corresponding to the midpoint bucket. Once the split key is determined, the original data node is partitioned to create two balanced child data nodes. This change was necessary because a linear inversion would not be very accurate under polynomial models, especially in non-linear regions where linear approximation could lead to poor partitioning and incidentally degrade performance. This update improves precision and correctness in splitting decisions and retains efficiency when integrated with existing data node and parent model logic.

4 EXPERIMENTAL RESULTS

We evaluate the performance of our Poly-ALEX extension using synthetic datasets containing 100k, 200k, 300k, 400k and 500k records. The benchmark tests batch lookup performance, with queries randomly distributed such that 75% of the keys are present in the dataset and 25% are absent. This setup helps simulate a realistic and diverse workload to assess performance under mixed query outcomes. We measure both index creation time and lookup throughput across dataset sizes to evaluate the scalability and efficiency of the extension.

Figure 2 illustrates the index creation time for different models—Linear, Polynomial, and ART—across increasing dataset sizes. It is observed that ART index takes longer to build than learned indexes. This is because ART involves recursive node allocations and pointer-based tree construction, which introduces significant memory management overhead. The Polynomial model requires slightly more time than the Linear model (around 1.1x to 1.4x that of the Linear model) due to the added computational complexity

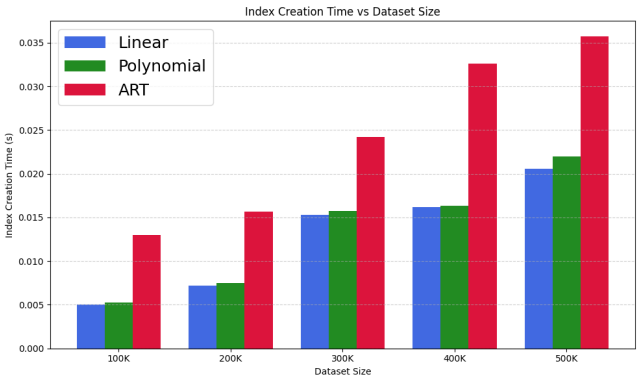

Figure 2: Total time taken (in seconds) by Linear, Polynomial and ART indexes for index creation.

of solving a system of equations using Cramer's Rule for fitting a second-degree polynomial.

Similarly, *Figure 3* presents the lookup throughput of different indexes across increasing dataset sizes. Initially, the ART index demonstrates higher throughput due to its fast pointer-based tree traversal, which excels on smaller datasets with minimal cache misses. At smaller scales, the Linear model also performs well due to its simple arithmetic prediction and low memory overhead. However, as the dataset size increases, the Polynomial model achieves up to 1.5x higher throughput than than Linear and ART indexes. This is because Polynomial model is able to get more accurate key position predictions, reducing the number of corrections needed after the initial model guess, thus optimizing performance at scale.

Figure 4 compares the model sizes in Megabytes (MB) of Linear, Polynomial, and ART indexes across increasing dataset sizes. The ART index consistently consumes more memory up to 1.2x larger than learned indexes, primarily due to its pointer-intensive structure and internal node metadata, which scale with the number of keys. In contrast, the Linear and Polynomial models exhibit nearly identical sizes. This is expected, as the Polynomial model introduces only an additional coefficient for its quadratic term, resulting in minimal storage overhead while shifting most complexity to runtime computations at index creation time.

The experimental results highlight the trade-offs between traditional and learned index structures across various performance metrics. While the ART index offers strong lookup performance on smaller datasets due to its pointer-based traversal, it suffers from higher index creation time and significantly larger memory footprint. On the other hand, the Linear model is efficient to build and compact in size but cannot capture complex, non-linear ddata distributions. The Polynomial model, though slightly slower to construct, demonstrates superior throughput on larger datasets, striking a balance between predictive accuracy and resource efficiency. These findings indicate that learned indexes, particularly polynomial-based ones or in-fact, more complex Machine Learning approaches, can be a compelling alternative to traditional structures like ART.

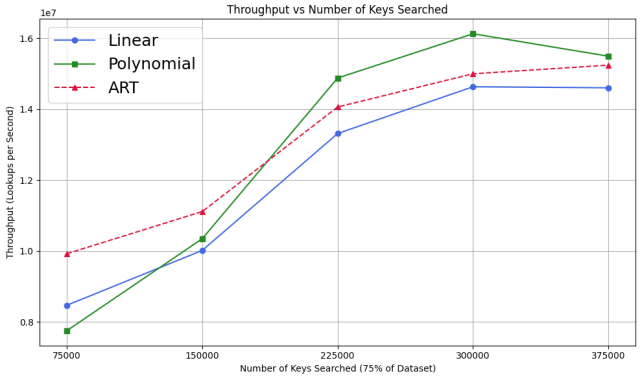

Figure 3: Throughput (in keys/second) by different indexes vs Number of lookups by OLAP queries

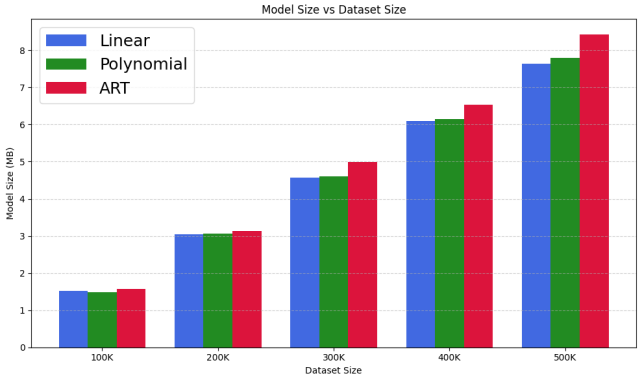

Figure 4: Model size (in MB) used by different indexes for different datasets

5 FUTURE WORK

While our implementation is able to get better results on tested workloads as compared to traditional ALEX and ART, there are certain areas where further development and optimization are needed.

One possible optimization is to integrate ALEX directly into the DuckDB source code, which would allow more accurate performance evaluation by eliminating the overhead of the extension interface.

To ensure the polynomial regression system scales effectively with large datasets and remains robust across diverse distributions, several enhancements can be made:

- **Numerical Stability via Input Normalization**: Normalize or standardize input keys (e.g., subtract mean and divide by standard deviation) to prevent overflow and loss of precision when computing higher-degree terms.
- **Efficient Solvers**: Replace Cramer's Rule with more scalable numerical methods such as LU decomposition [2] or QR factorization [1], which offer better performance and numerical robustness on large data batches.
- **Adaptive Degree Selection**: Dynamically select between linear, quadratic, or higher-degree models based on data variance, error metrics, or cost models—ensuring minimal model complexity without sacrificing accuracy.
- **Memory-Efficient Aggregation**: Implement online algorithms or sketching techniques (e.g., Welford's method [5]) for computing moments and aggregates incrementally to handle streaming data or large index partitions.

6 CONCLUSION

In this project, we extended ALEX, a learned index structure, by implementing polynomial models and integrated Poly-ALEX into DuckDB. Our goal was to overcome the drawbacks of linear models on non-linear key distributions, which are common in real-world workloads. We introduced extensive structural modifications across bulk loading, cost sampling, root expansion, downward splitting, and data node splitting routines. These changes ensured that polynomial models could be used throughout the index with correctness and efficiency.

Our experiments show that Poly-ALEX achieves significant improvements over the linear ALEX indexes and DuckDB's ART indexes. Polynomial models introduce an increase in index creation time (around 1.1x to 1.4x as compared to the linear model) and minimal additional memory overhead, but they make up for this by offering up to a 1.5x higher lookup throughput on large datasets. Importantly, model size remains similar to that of the linear models, and has significantly less memory overhead than ART.

Overall, these results show the promise of higher order models for OLAP-style workloads. Further enhancements are necessary for broader applicability, including numerical stability improvements, more scalable solvers for polynomial fitting, and adaptive model complexity selection.

6.1 Acknowledgments

We would like to express our gratitude to the ALEX (Adaptive Learned Indexes) [3] and RMI (Recursive Model Indexes) [8] research teams for their groundbreaking work in learned index structures. Their innovative approaches to data indexing have significantly influenced our research direction and provided valuable insights that helped shape the methods presented in this paper. The open-source implementations and detailed publications from these projects were instrumental in our comparative analysis and the development of our own techniques.

REFERENCES

[1] E Anderson, Zhaojun Bai, and J Dongarra. 1992. Generalized QR factorization and its applications. *Linear Algebra Appl.* 162 (1992), 243–271.
[2] Xilin Cheng. 2023. Comparative Analysis of Linear Regression, Gaussian Elimination, and LU Decomposition for CT Real Estate Purchase Decisions. *arXiv preprint arXiv:2311.13471* (2023).
[3] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 969–984.
[4] DuckDB. [n.d.]. GitHub - duckdb/duckdb: DuckDB is an analytical in-process SQL database management system — github.com. https://github.com/duckdb/duckdb.
[5] Andrey A Efanov, Sergey A Ivliev, and Alexey G Shagraev. 2021. Welford's algorithm for weighted statistics. In *2021 3rd International Youth Conference on Radio Electronics, Electrical and Power Engineering (REEPE)*. IEEE, 1–5.
[6] DuckDB Extensions. [n.d.]. GitHub - duckdb/extension-template: Template for DuckDB extensions to help you develop, test and deploy a custom extension — github.com. https://github.com/duckdb/extension-template.

[7] Kevin P. Gaffney, Martin Prammer, Larry Brasfield, D. Richard Hipp, Dan Kennedy, and Jignesh M. Patel. 2022. SQLite: past, present, and future. *Proc. VLDB Endow.* 15, 12 (Aug. 2022), 3535–3547. https://doi.org/10.14778/3554821.3554842

[8] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*. 489–504.

[9] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 185–196. https://doi.org/10.1109/ICDE.2018.00026

[10] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[11] Microsoft. [n.d.]. GitHub - microsoft/ALEX: A library for building an in-memory, Adaptive Learned indEX — github.com. https://github.com/microsoft/ALEX.

[12] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 international conference on management of data*. 1981–1984.

[13] Stephen M Robinson. 1970. A short proof of Cramer's rule. *Mathematics Magazine* 43, 2 (1970), 94–95.

[14] Mohamed Ziauddin, Andrew Witkowski, You Jung Kim, Dmitry Potapov, Janaki Lahorani, and Murali Krishna. 2017. Dimensions based data clustering and zone maps. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1622–1633. https://doi.org/10.14778/3137765.3137769