Learned Indexes in DuckDB

Venkataraman Venkatasubramaniam
University of Southern California
Los Angeles, California, US
vv70613@usc.edu

Mayur Prasanna
University of Southern California
Los Angeles, California, US
mayurpra@usc.edu

Siddharth Raj Dash
University of Southern California
Los Angeles, California, US
srdash@usc.edu

ABSTRACT

Traditional index structures such as B-Trees and Adaptive Radix Trees (ART) provide reliable general-purpose performance but incur high memory overhead and frequent CPU cache stalls due to pointer-heavy traversal. To explore a more memory-efficient alternative, in this project, we integrate Learned Indexes into DuckDB using a Recursive Model Index (RMI) that models index key lookups as a regression problem. Our extension uses a static model for index prediction within a read-only table together with a dynamic overflow structure to support writes, enabling fast and computation-driven lookups. Evaluated against DuckDB's native ART index on synthetic datasets, our implementation reduces index memory usage by approximately 60% while maintaining comparable query latencies.

1 INTRODUCTION

Modern database systems rely heavily on index structures to support fast analytical and transactional workloads. For decades, B-Trees and their variants, such as DuckDB's Adaptive Radix Tree (ART), have served as the default choice for general-purpose indexing. Their deterministic performance, balanced search guarantees, and mature implementations make them robust under a wide variety of data distributions. However, as datasets grow into the billions of records and memory bandwidth becomes a limiting factor, traditional tree structures reveal several inefficiencies. The heavy use of pointers, multi-level node traversal, and random access patterns increasingly mismatch the behavior of modern CPU architectures, where computation is cheap but memory access is expensive.

The resulting performance characteristics present a structural challenge: even well-engineered tree indexes incur substantial memory consumption and frequent cache misses as lookups require navigating from root to leaf. Meanwhile, the distribution of index keys in many real-world datasets is not arbitrary; often it is structurally sorted, partially ordered, or statistically predictable. These observations motivate a natural question: if the relationship between index keys and their positions is predictable, do we really need to walk through a multi-level tree with multiple indirection pointers to find them? This shift in perspective opens the door for indexing methods that replace memory-bound traversal with computation-heavy prediction.

1.1 Problems with Traditional Indexing

Traditional indexing structures are optimized for worst-case generality rather than best-case specialization. B-Trees, for example, rely on wide internal nodes (fan-out) and careful node balancing, yet every lookup still requires navigating multiple levels of indirection via a sequence of comparisons and branching decisions. ART improves upon this by dynamically adapting node sizes to key diversity, but they remain rooted in the same tree traversal principles. Each pointer de-reference risks a potential CPU cache miss, and random memory accesses accumulate as the index trees grow deeper with increasing data size.

These characteristics lead to several well-known limitations. (1) Memory overhead becomes a dominant concern with internal nodes, child pointers, and metadata consuming upto 30–40% of the total index footprint. (2) The cost of pointer chasing increases with dataset size as these index tree structures gain height, resulting in more unpredictable memory jumps and leading to poor scalability. (3) These tree indexes make no attempt to exploit the distribution of the data. Whether the underlying key space is uniform, monotonic or skewed, a tree generates the same navigational pattern.

These factors create a widening gap between what traditional trees offer and what modern CPU hardware delivers.

1.2 Learned Indexes

Traditional indexes assume that lookup must be performed by traversing a search structure. Learned indexes challenge this assumption by observing a key insight that if data is structurally ordered, it implicitly defines a monotonic function:

$$F(k) = \texttt{position of key k}$$

The mapping from a key to its physical position closely resembles the cumulative distribution function (CDF) of the keys. If this mapping is smooth, then a model predicting this function can approximate it far more compactly than a tree. Instead of navigating downward through a hierarchy of nodes, a learned index's lookup operation can directly estimate the position using a predictive model.

$$\hat{p} = M(k)$$

where M is a predictive model—linear, polynomial, neural, or a combination model.

For example, a simple regression model can approximate the mapping between keys and their physical array indices. Because model evaluation requires only a handful of arithmetic operations, the cost is closer to *O(1)* than the *O(log N)* behaviour of trees.

Once a predicted position is produced, the system performs a small refinement scan around that estimate to guarantee correctness. This hybrid strategy replaces long chains of memory accesses with a short prediction and a narrow, cache-friendly search window.

The benefits are significant: memory overhead drops sharply because most internal tree data structures are eliminated, CPU cache locality improves, and point lookups or short-range queries execute with fewer instructions.

1.3 Base Paper: Brief Overview

The foundational paper "The Case for Learned Index Structures"[1] re-frames database indexing as a function approximation problem. Instead of viewing indexes as navigational trees, the authors argue that the correct mental model is a mathematical function mapping keys to positions. In many datasets, this function is smooth enough to be approximated by small, lightweight models. The paper demonstrates that such models can replace or augment traditional data structures, achieving lower memory usage and faster lookup times while preserving correctness through controlled error bounds.

2 IMPLEMENTATION

Our system is implemented as a fully functional learned index inside DuckDB following its loadable extension architecture. The extension registers a new index type (`RMI`), that can be dynamically loaded into DuckDB at runtime via the `RMIModule`, or compiled directly into the DuckDB binary during the build process.

At a high level, the extension introduces:

- a new index structure with model-based prediction,
- multiple model families (linear, polynomial, two-layer RMI) selectable using create index options,
- a custom physical plan operator (`rmi_index_scan`) that performs prediction, bounded refinement, and overflow merging, and
- debugging PRAGMAs that expose the internal learned index state.

This implementation strategy preserves DuckDB's optimizer, storage engine, and execution model while inserting RMI behavior at well-defined extension hooks. This can be better understood through two workflows that drive the entire system. (1) The SQL `CREATE INDEX`, which triggers index construction, and (2) The SQL `SELECT`, which triggers index lookup and model-guided scanning.

2.1 Index Creation Flow

Index creation is driven by the `CREATE INDEX` statement and proceeds through planner, physical create, and model training stages.

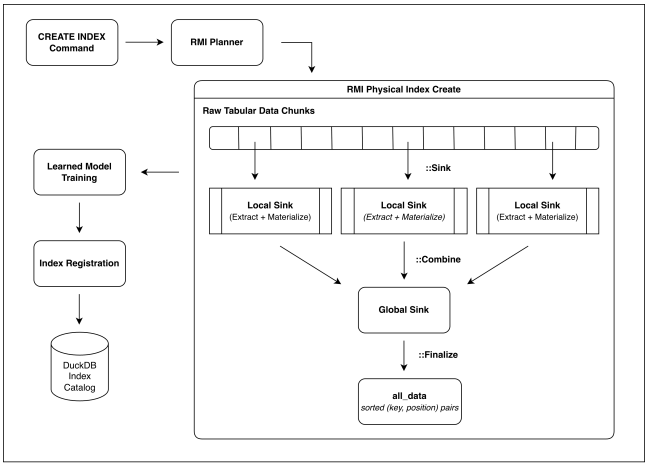

Figure 1: RMI Index Creation Flow

Index Creation. DuckDB exposes the RMI index through the standard SQL interface via the `CREATE INDEX` command as follows:

```
CREATE INDEX idx_rmi ON t USING RMI(col)
    WITH (model = 'linear' | 'poly' | 'two_layer');
```

The index currently supports only numeric types as model fitting requires ordered, continuous domains. The column must also be `NOT NULL`, and the binder explicitly validates these constraints before the index can be constructed.

Planner and Physical Create. When the user issues `CREATE INDEX ... USING rmi`, DuckDB's planner invokes the extension's binder which first validates index eligibility (numeric type, NOT NULL column, supported options etc.), followed by emitting a custom physical operator (the *PhysicalCreateRMIIndex*). This operator is inserted into the execution pipeline and materializes all keys required to build the index.

Parallel Sink and Combine (Source–Sink). DuckDB executes the pipeline with multiple worker threads. Each worker receives vector-sized chunks of the indexed column and accumulates keys and row identifier pairs into a thread-local Sink. When workers complete, the `Combine()` phase merges local buffers into a single Global Sink State. As DuckDB stores data in hybrid-columnar vectors, we flatten the collected tuples into a contiguous `std::vector<RMIEntry>` of `(key, rowid)` pairs to enable efficient sorting and model training.

Finalize and Model Training. After `Combine()`, the `Finalize()` stage performs a strict sequential sort over the flattened (key–rowid) pairs. The sorted keys and their positions are supplied to `RMIIndex::Build`, which computes model parameters depending on the chosen model family (linear, polynomial, or two-level RMI). In addition, each model also computes error bounds that are used during query-time refinement. The resulting physical index contains only the sorted array, trained model parameters, error bounds and an overflow map, keeping the index footprint minimal.

Catalog Registration. Once built, the index is registered with DuckDB via *table.GetStorage().AddIndex()*, making it available to the optimizer.

Overflow Region and Write Support. As retraining the model for each write would be expensive, we propose a hybrid layout: a static sorted region covered by the trained model and a dynamic overflow region backed by an ordered map. New inserts are appended to the overflow map without disturbing the base model. During lookups, the scanner queries both regions, merging the overflow results with model-guided results to produce a complete and consistent response. Periodic or manual re-indexing can fold the overflow region back into the base model.

2.2 Index Scan Flow

Index scans over RMI-backed columns replaces DuckDB's default sequential or ART-based lookup path. Once selected by the optimizer, execution proceeds through operator binding, model-based prediction, and predicate-specific refinement, followed by vectorized tuple retrieval.

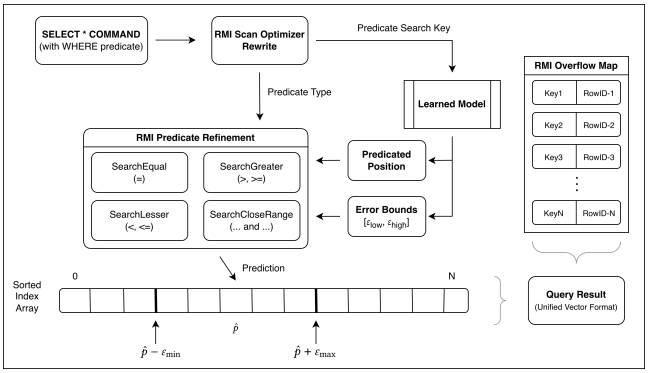

Figure 2: RMI Index Scan Flow

Optimizer Rewrite and Scan Selection. During planning, the extension introduces a rewrite rule that inspects `WHERE`-clause predicates. If a predicate references a column with an attached RMI index and the predicate is one of the supported forms (equality, one-sided inequality, or two-sided range), the optimizer rewrites the logical `GET` into a custom `RMIIndexScan` operator. This preserves projection and filtering semantics but shifts key lookup to the learned model rather than to a tree traversal.

Learned Model Lookup. At execution time, the scanner extracts the bound predicate key(s) and calls the trained model to obtain an estimated position:

$$\hat{p} = \text{model.predict}(k).$$

The model's learned error window $[\varepsilon_{\min}, \varepsilon_{\max}]$ is used to compute a safe search interval via

$$[p_{\text{low}},\ p_{\text{high}}] = \text{model.GetSearchBounds}().$$

By jumping to this predicted region, the scanner reduces the search space from millions of rows to a small, model-guided bounded interval.

Refinement and Predicate Dispatch. After determining the model-derived bounds, the scanner performs a local linear scan to locate predicate matches. The predicate-specific refinement is done as follows:

- **SearchEqual**: scan the model-guided window and collect RowIDs whose keys match exactly.
- **SearchGreater** / **SearchGreaterEqual**: start at $\hat{p} - \varepsilon_{\min}$ and scan forward to the end of the array.
- **SearchLess** / **SearchLessEqual**: scan from the beginning of the array up to $\hat{p} + \varepsilon_{\max}$.
- **SearchCloseRange**: compute model windows for both predicate endpoints and scan the intersected interval.

The main sorted index array provides deterministic ordering and results are collected into an ordered RowID buffer. Newly inserted records in the overflow map are also scanned, evaluated and merged into this buffer.

Vectorized Fetch. After refinement, RowIDs from both the static sorted region and the overflow map are merged into a unified, result vector. These RowIDs are materialized into DuckDB's *UnifiedVectorFormat*, and a vectorized `Fetch` call retrieves the payload tuples from table storage.

2.3 Models in Detail

Our RMI extension supports three regression-based model families to map keys to positions. To ensure numerical stability for datasets over large key ranges, we employ mean-centering and variance reduction techniques during training.

Linear Regression RMI. The Linear RMI approximates the cumulative distribution of sorted keys using a single-dimensional linear regressor

$$\hat{y} = m(x - \mu_x) + c$$

where μ_x is the mean of the key distribution. Mean-centering prevents loss of precision that occurs when datasets have large absolute key values. Because this model maintains only two floating-point parameters, it has negligible memory overhead and $O(1)$ prediction cost. It performs well when the sorted keys follow an approximately linear trend, such as uniformly distributed or monotonic integer sequences. However, it cannot capture curvature or local nonlinearities, which leads to wider error bounds in skewed datasets.

Polynomial Regression RMI. The Polynomial Regression RMI generalizes the linear model with higher-degree terms to capture non-linear key distributions.

$$\hat{y} = a_0 + a_1(x - \mu_x) + a_2(x - \mu_x)^2 + \cdots + a_d(x - \mu_x)^d.$$

We evaluate degrees from 1 to a configurable maximum (default $d = 6$), selecting the model that minimizes mean-squared error while remaining numerically stable. Polynomial models significantly reduce error windows in datasets with curvature or piecewise trends, but incur higher computational cost during both training and inference.

Two-Level RMI. The two-level RMI captures both global and local structure by decomposing the key space hierarchically. A top-level parent linear model predicts a segment index:

$$\hat{s} = m_p(x - \mu_x) + c_p,$$

and the key is routed to one of $K = \lfloor \sqrt{N} \rfloor$ child segments. Each child segment trains its own local linear regressor using mean-centered keys within the segment:

$$\hat{y} = m_c(x - \mu_{x,c}) + c_c.$$

This structure reflects the insight that although the global CDF may contain curvature, local fragments are nearly linear. The two-level RMI therefore achieves much tighter error bounds than a single global regressor while keeping model size small (only K child models). During lookup, the system probes segment $\hat{s}$ and adjacent segments ($\hat{s} \pm 1$) to maintain correctness guarantees.

2.4 Debugging via Custom PRAGMAs

Learned indexes act as model-driven structures, so we expose PRAGMA-style utilities to inspect training state, error bounds, and runtime behavior otherwise hidden inside the "black box".

`pragma_rmi_index_info`. Provides a one-line catalog overview of all registered RMI indexes and verifies that each has a valid physical instance.

`rmi_index_dump(name)`. Returns the internal sorted key–rowid array used for training, enabling quick checks of ordering correctness and input-domain validity.

`rmi_index_model_stats(name)`. Reports per-key predicted positions and global error bounds to assess model accuracy and detect underfitting or overly wide windows.

`rmi_index_overflow(name)`. Displays inserted keys stored in the overflow region, serving as an indicator of when retraining or model refresh may be required.

`rmi_index_model_info(name)`. Summarizes the trained model by reporting family type, global error bounds, overflow size, and relevant parameters for linear, polynomial, or two-layer RMI models.

Together, these PRAGMAs provide a compact introspection toolkit that makes learned indexes transparent and debuggable throughout development and evaluation.

3 EXPERIMENTS AND RESULTS

3.1 Experimental Setup

Benchmarking Environment. All experiments were conducted inside an isolated dockerized container running a custom DuckDB build with the RMI extension. The container was provisioned with 8GB of container memory and 2 vCPUs to ensure consistent and reproducible performance across runs. This controlled environment eliminated variance arising from host processes, background I/O noise or dynamic resource allocations.

Dataset. To evaluate the behavior of our learned index under diverse workloads, we generated synthetic numerical datasets exhibiting a range of statistical characteristics. These included uniformly distributed data (capturing near-linear patterns), polynomial trends with smooth curvature, and randomized or skewed distributions (to stress-test non-linear model behavior). For every dataset, we validated correctness by comparing all query results against full-table scans.

Corresponding to increasing data scale, we evaluated three dataset sizes across 10^3 tuples, 10^4 tuples and 10^5 tuples. To evaluate query performance, we constructed a mixed query workload consisting of both point lookups and short-range predicates. Point queries tested the model's precision, while small range filters utilized the model-derived search windows and refinement logic. Over 100 such queries across the dataset were part of the evaluation script. Together, these queries approximate real-world analytical and OLTP-style access patterns found in modern workloads.

Metrics. To get a holistic evaluation report of our implementation, we measure the following metrics:

- **Index Memory Footprint**: The total memory consumed by our learned indexes after successful creation.
- **Index creation time**: End-to-end latency of building the index, including key materialization, sorting and model training.
- **Query Execution Time**: The time taken to execute each of the queries in the query set.
- **Hit Rate**: Percentage of queries that successfully executed and extracted the correct tuples.

These metrics collectively capture efficiency (query time), scalability (construction time), memory behavior (footprint), and correctness (accuracy), enabling a grounded comparison against DuckDB's built-in Adaptive Radix Tree (ART) index across a range of data distributions.

3.2 Results and Observations

RMI Debugging PRAGMAs. To inspect the correctness of the index structures and the behavior of the learned models, we developed several auxiliary PRAGMAs. These allow users to visualize sorted key–rowid arrays, model parameters, segment structure, and overflow contents. Figure 3 below illustrate the outputs of the monitoring PRAGMAs for a sample index.

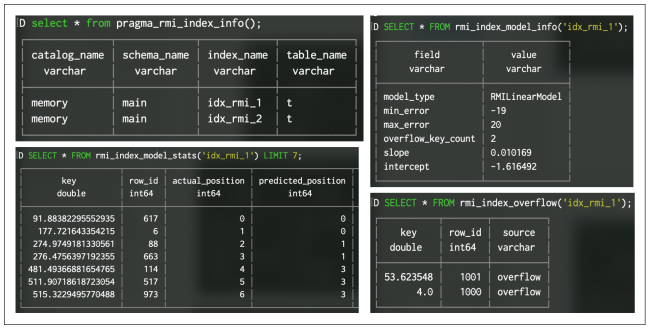

Figure 3: Learned Index PRAGMAs

Results. The following tables and graphs summarize the performance metrics of the learned index against the ART index, averaged across the dataset sizes.

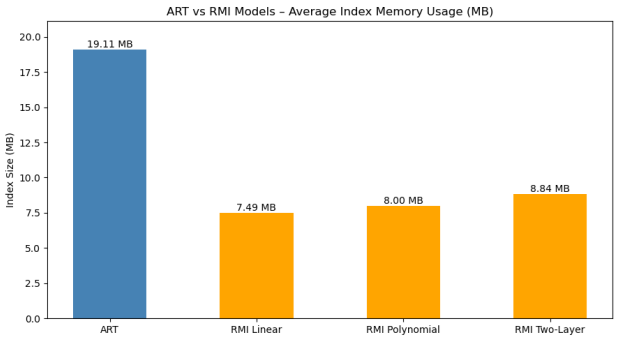

Figure 4: Comparisons of metrics between RMI models and ART index.

Observations. Our experiments reveal several notable characteristics of the proposed learned RMI index.

Metric	ART	Linear RMI	Polynomial RMI	Two-level RMI
Memory footprint (MB)	19.11	7.49	8.00	8.84
Query Execution Time (ms)	561.34	550.19	546.43	555.24
Index Creation Time (ms)	2746.67	2770.00	2880.00	3026.67
Hit Rate (%)	100	100	100	100

Table 1: Comparison of Learned Index Variants Against DuckDB ART Baseline

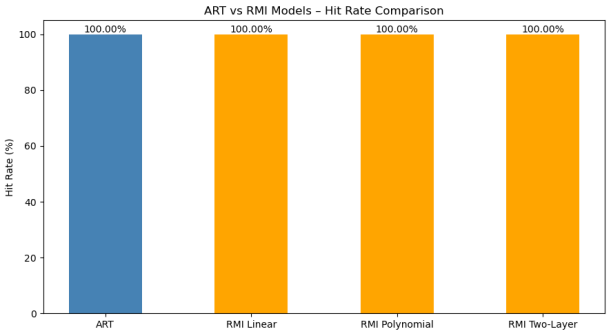

Figure 5: Comparisons of metrics between RMI models and ART index.

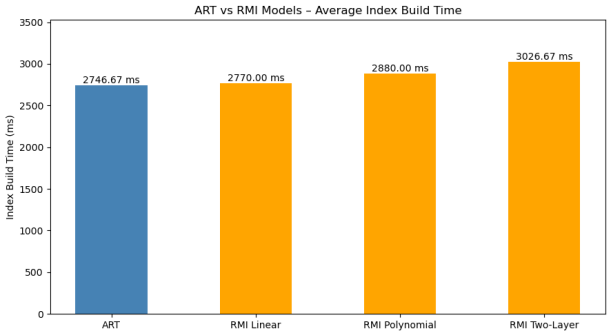

Figure 6: Comparisons of metrics between RMI models and ART index.

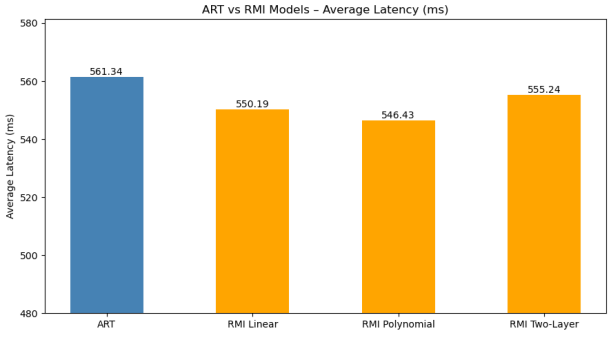

Figure 7: Comparisons of metrics between RMI models and ART index.

- **Lightweight Index**: The learned indexes use a significantly lower memory footprint. Compared to ART it is uses nearly 60% less memory as a learned index only needs to store the model parameters instead of an entire radix tree, allowing for better utilization of the residual space for DuckDB's internal operations.
- **Comparable Execution time**: Across the benchmarks we noticed that the queries take similar run times to complete execution. In most cases the learned indexes perform marginally better with a 15-20 ms reduction in execution time, allowing for better throughput across larger workloads.
- **Slightly higher index creation time**: This is a tradeoff of using learned indexes. Our models take slightly longer to create the index because of the underlying sorting of the data and model training, however the overhead is only marginally higher than that of the standard ART index and since index creation is only a one-time process done in the background it's impact is not felt across a workload. We can further reduce this overhead by implementing parallel model training that would significantly reduce the index creation time.
- **High robustness and accuracy**: Our proposed learned indexes have a 100% hit rate, making them as reliable as the standard DuckDB's ART index. This proves that learned indexes can be a trustworthy addition to DuckDB's indexing catalog.

4 CONCLUSION AND FUTURE WORK

Conclusion. Learned indexes in DuckDB provide a practical, memory-efficient alternative to traditional structures, reducing index footprint by over 60% through compact model parameters while still offering competitive lookup performance. Although index construction involves sorting and model training, this cost is one-time and can be reduced through parallelism; once built, the RMI consistently delivers accurate predictions with a 100% hit-rate on evaluated workloads.

Future Work. Looking forward, several directions could enhance the performance and applicability of Learned Indexes. (1) Adaptive segmentation strategies could allow the index to better handle non-uniform key distributions in the two-layer model. (2) Paging support for large outputs could improve memory and I/O efficiency for massive result sets. (3) Exploring deep learning–based learned models could capture highly complex distributions, trading increased computation for improved accuracy and flexibility.

REFERENCES

[1] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data (SIGMOD '18)*, pages 489–504. DOI: https://doi.org/10.1145/3183713.3196909.

[2] Ryan Marcus, Emily Zhang, and Tim Kraska. CDFShop: Exploring and Optimizing Learned Index Structures. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*, pages 2789–2792. DOI: https://doi.org/10.1145/3318464.3384706. GitHub Repository: https://github.com/learnedsystems/RMI.

[3] DuckDB-VSS: Vector Similarity Search Extension for DuckDB. GitHub Repository: https://github.com/duckdb/duckdb-vss/.

[4] DuckDB-Spatial: Spatial Indexing and Geometry Extension for DuckDB. GitHub Repository: https://github.com/duckdb/duckdb-spatial/tree/main.

[5] DuckDB Documentation. Extending DuckDB with Loadable Extensions. https://duckdb.org/docs/extensions/overview.