USC Viterbi School of Engineering

Learned Indexes in DuckDB

Team members:

Venkataraman (9535346631),

Mayur Prasanna (3182622085),

Siddharth Raj Dash (5854395979)

Overview

USC Viterbi School of Engineering

- ❏ Introduction
- ❏ Learned Indices
- ❏ RMI Architecture
- ❏ Implementation
- ❏ Results
- ❏ Observations

The Problem

The Standard: B-Trees (and DuckDB's ART) are the gold standard for general-purpose database indexing.
The Cost of "General Purpose":

- **Memory Overhead:** Trees require storing massive internal nodes and pointers, often consuming 30-40% of the total database size.
- **Cache Misses:** Traversing a tree involves "pointer chasing" across random memory pages, which stalls the CPU.
- **Scalability:** As data grows, tree height increases, adding more distinct memory jumps for every lookup.

The Opportunity: If we know the data distribution (e.g., sorted integers), why do we need to traverse a tree to find a value?

USC Viterbi School of Engineering

Why Learned Indexes?

- Index lookup can be modeled as a **regression problem**
- If data is sorted, model can estimate:
  key → approximate position
- Enables:
  - Fewer instructions than B-Tree traversal
  - Better CPU cache friendliness
  - Reduced memory overhead
- Goal: *Accelerate point lookups and short range queries and reduce seek time*

Core Concept: Indexing is actually a "Cumulative Distribution Function" (CDF) problem.

- We want to map a **Key** to a sorted **Position**.

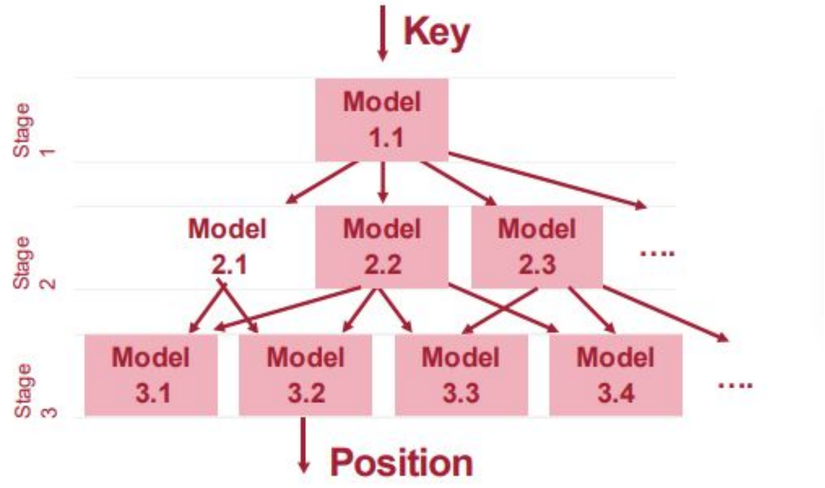
The Mechanism:

- Instead of if node < x then go left, we calculate position = slope * x + intercept.

The RMI Architecture:

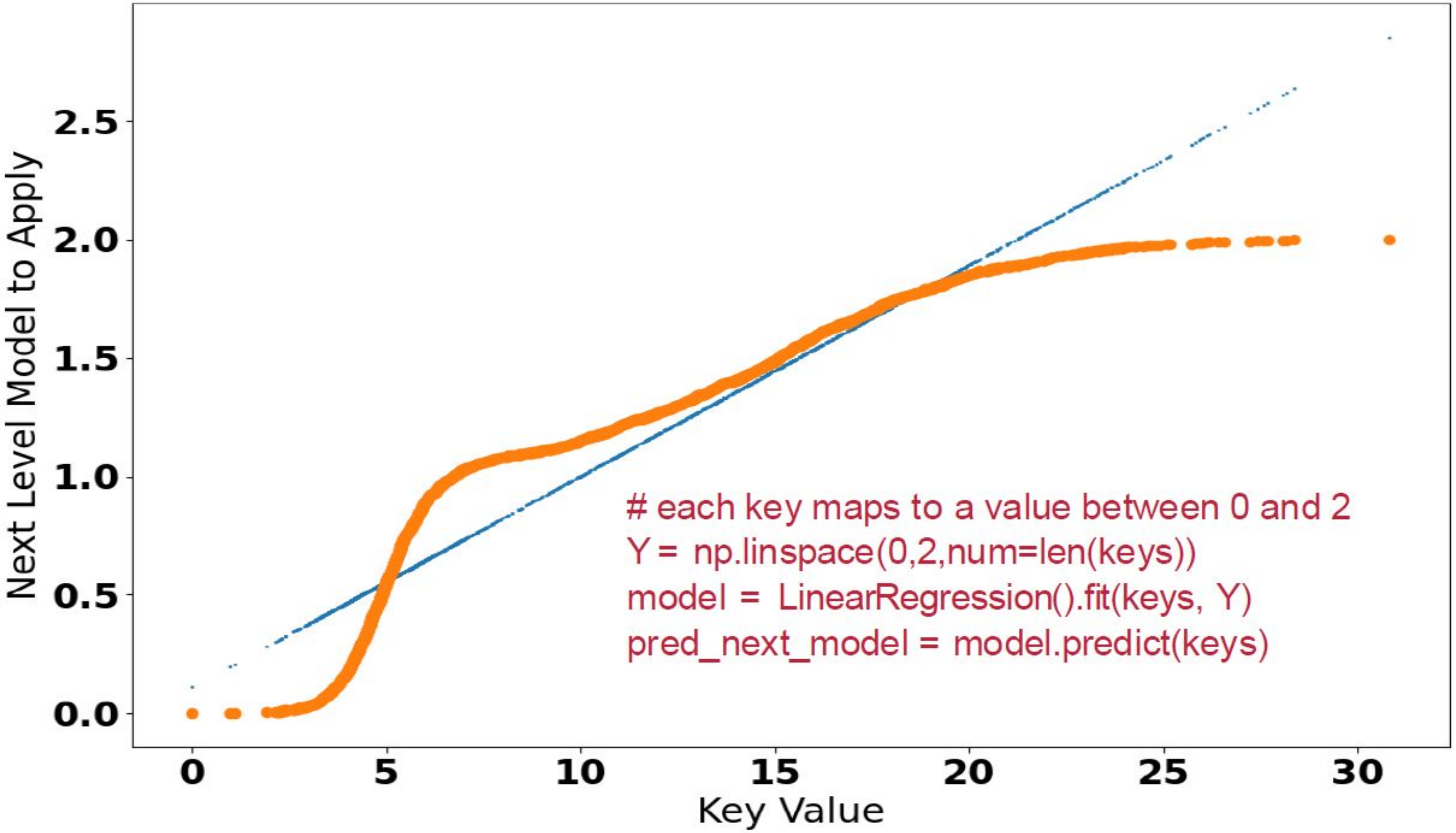
- **Model:** A lightweight Regression model (or Neural Net) approximates the data location.
- **Hierarchy:** Can be recursive (Model A picks Model B), though our implementation focuses on a single-stage linear model for simplicity.

Key Benefit: We trade **Memory Accesses** (slow) for **CPU Arithmetic** (fast).



Learned Indices example

An example with a linear prediction model



Learned Indices Implementation

USC Viterbi School of Engineering

Models Used

- **Linear Regression RMI**
  - Simple, fast, minimal overhead
- **Polynomial Regression RMI**
  - Higher capacity, handles curvature
- **Two Level RMI**
  - Best accuracy via intentional overfitting
  - Chosen as final model for benchmarking

System Architecture & Integration

USC Viterbi School of Engineering

Platform: DuckDB (C++ columnar analytical database).

Implementation Strategy: Built as a "Loadable Extension."

- Does not modify DuckDB core source code.
- Uses DuckDB's **Table Function** API to define custom scanning logic.

Components:

- **Planner:** Intercepts CREATE INDEX to build our custom plan.
- **Builder:** Materializes data into a dense, sorted format.
- **Scanner:** Executes the model prediction during SELECT queries.
- **Index Core:** A hybrid engine that manages the RMI Model for fast, efficient reads

Index Physical Create (Source-Sink Model)

The Sink Phase (Parallel Collection):

- DuckDB executes pipelines in parallel threads. Our operator acts as the "Sink" (the destination).
- **Method:** Sink() collects data chunks into thread-local buffers (LocalState).
- **Method:** Combine() merges these local buffers into one global list (GlobalState) as threads finish.

The Finalize Phase (The "Stop-the-World" Build):

- Once all data is collected, Finalize() takes over.
- **Sort:** We extract all keys and strictly sort them (Sequential Step).
- **Train:** We pass the sorted data to RMIIndex::Build to calculate the model weights.

Catalog Registration:

- After the model is trained, we call table.GetStorage().AddIndex().
- This tells DuckDB: "The index is ready. Future queries can use this pointer."

Index Execution

USC Viterbi School of Engineering

1. Getting the Search Key:

- When you run WHERE key = 100, the database engine extracts that raw number (100) and passes it to our scanner.

2. The "Math" Lookup:

- We plug 100 into our trained formula. It instantly calculates the memory address (Row ID).
- *Note:* We also quickly check the "Overflow List" here to catch any data added since the last build.

3. The Vector Handoff:

- We manually copy our list of IDs into the DuckDB vector format so the execution engine can understand them.

4. Fetching the Data:

- DuckDB takes these vectorized IDs and retrieves the full rows (the actual payload) from the disk storage.

Build Phase – From Data to Function

Step 1: Materialization:

- DuckDB stores data in vector chunks. We must flatten this into a contiguous memory block (vector of structs).

Step 2: Strict Sorting (The Critical Step):

- The model learns the position of keys. We utilize std::sort to prepare the training data.

Step 3: Model Training:

- **Input:** Pairs of (Key, Position)
- **Algorithm:** Simple Linear Regression, Polynomial Regression, 2 Level RMI.
- **Output:** Model Parameters: Slope and Intercept.
- **Footprint:** The "Index" is literally just the model weights plus the sorted data.

Scan Logic – The "Last Mile" Search

1. The Prediction (Global Jump)

- **Formula:** pos = model.predict(key)
- **Result:** This gives us a "Global Search Window" (e.g., Predicted Index +/- Max Error).
- **Efficiency:** Instantly narrows the search space from millions of rows to just a handful.

2. The Refinement (Local Linear Scan)

- **Action:** We perform a tiny linear scan *only* within that predicted window.

3. The Dispatcher (Handling Predicates)

Depending on the SQL operator, we switch execution to specialized methods:

SearchEqual (=):	SearchGreater (>, >=):	SearchLess (<, <=):
• Scans the predicted window for exact matches. • Optimized for Point Lookups; returns only the matching Row IDs.	• Locates the boundary key within the window. • Then scans **forward** to the end of the array to collect all subsequent rows.	• Locates the boundary key within the window. • Scans from the **start** of the array up to that boundary position.

Handling Writes – The Overflow Mechanism

The Constraint: RMI models are static. Retraining the regression model on every INSERT is prohibitively slow.

The Solution: A Hybrid Architecture.

- **Static Region:** The massive, sorted array managed by the RMI.
- **Dynamic Region:** A standard std::map<Key, RowID> (The Overflow).

Write Path: New rows go directly into the Overflow map.

Read Path: Queries must probe **both** the RMI and the Overflow map, merging results.

Experiments

USC Viterbi School of Engineering

Benchmark Environment:

- Isolated dockerized container with a C++ runtime environment.

Three levels:

- 1000 keys distribution
- 10000 keys distribution
- 100000 keys distribution

Query Types:

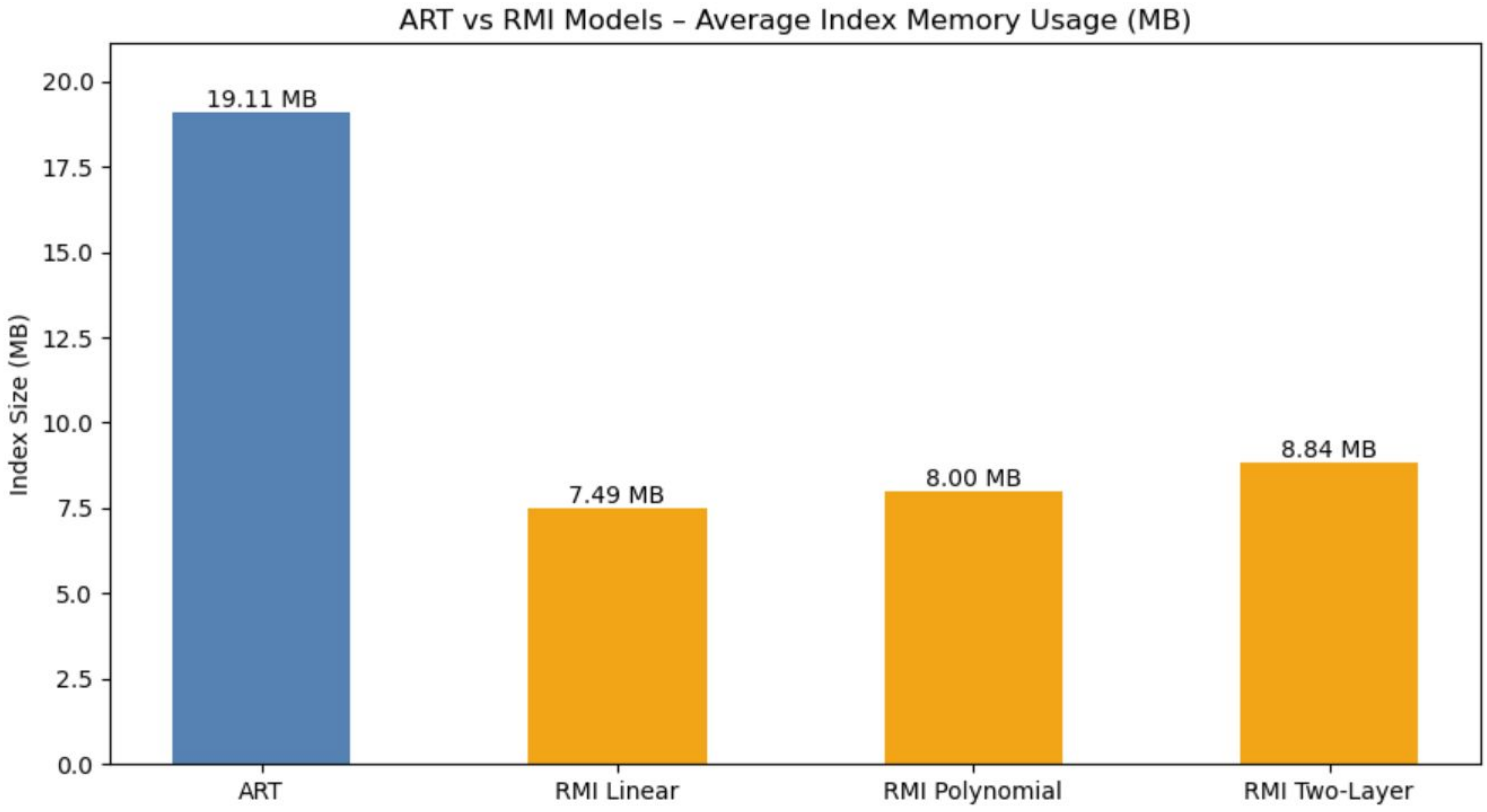
- Over 100 point lookup queries and small range search queries

Compared against:

- DuckDB ART Index
- RMI Index (Linear, Poly, Two Level RMI)

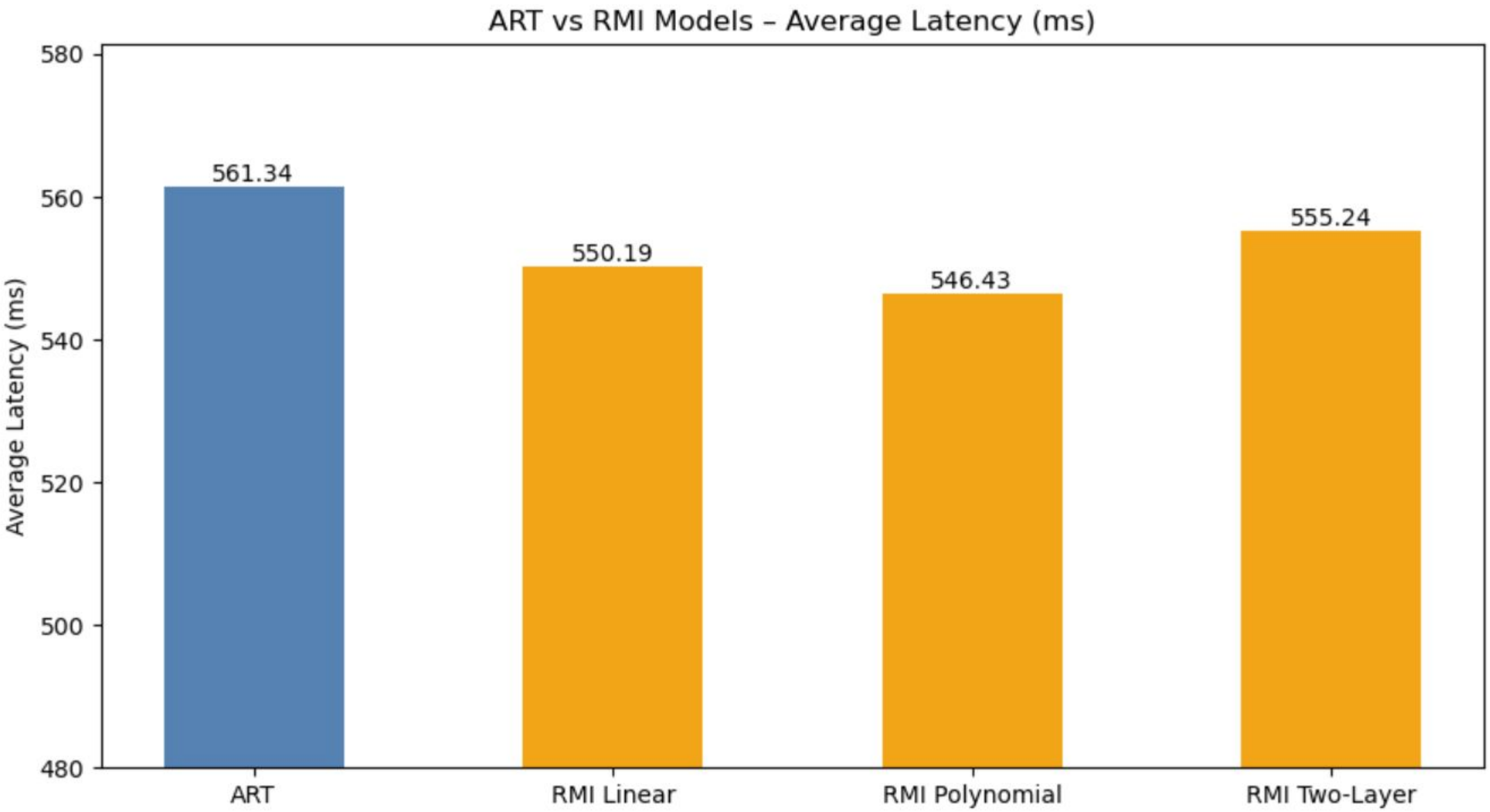
Results - Index Memory

USC Viterbi School of Engineering

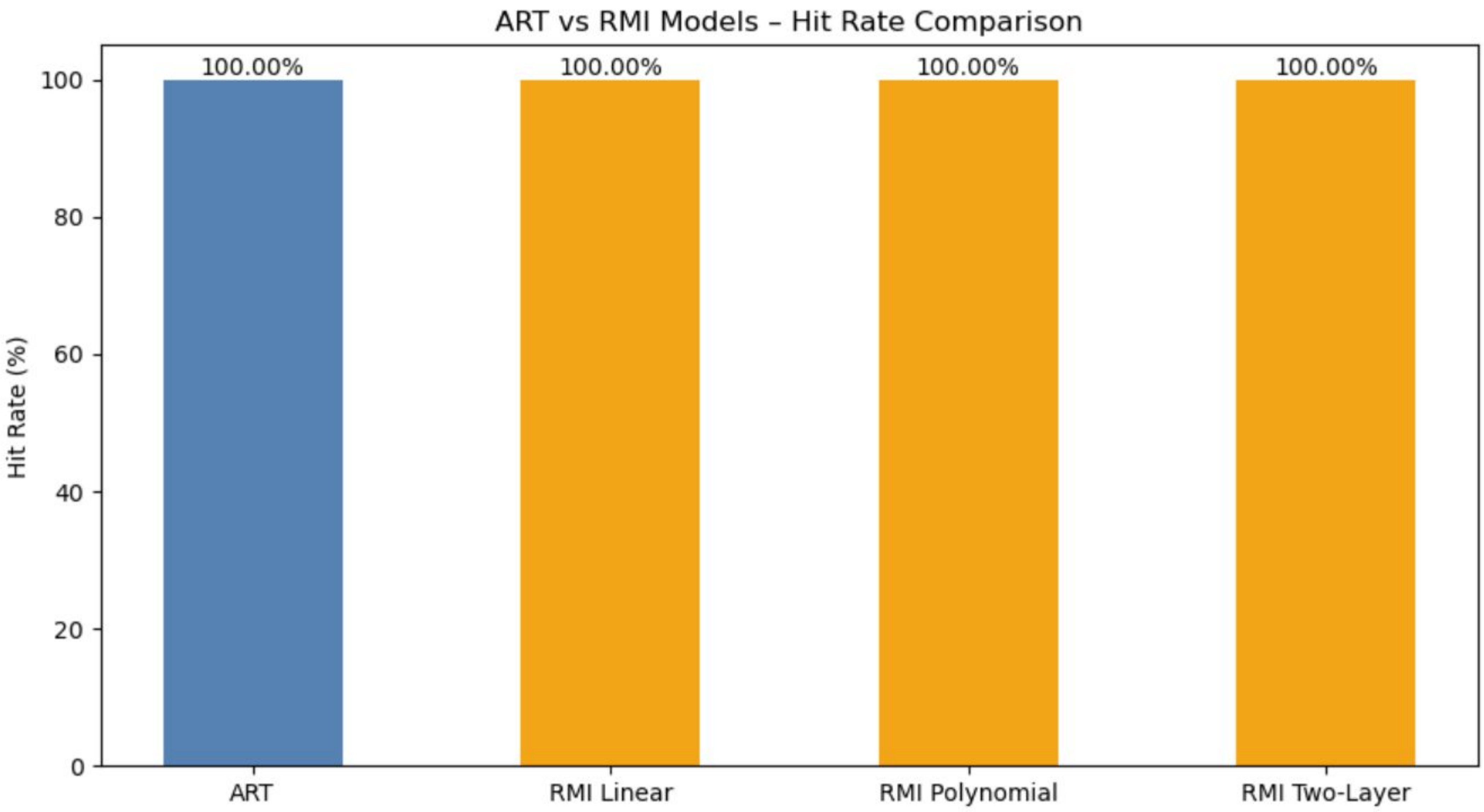


Results - Query Execution Time

USC Viterbi School of Engineering

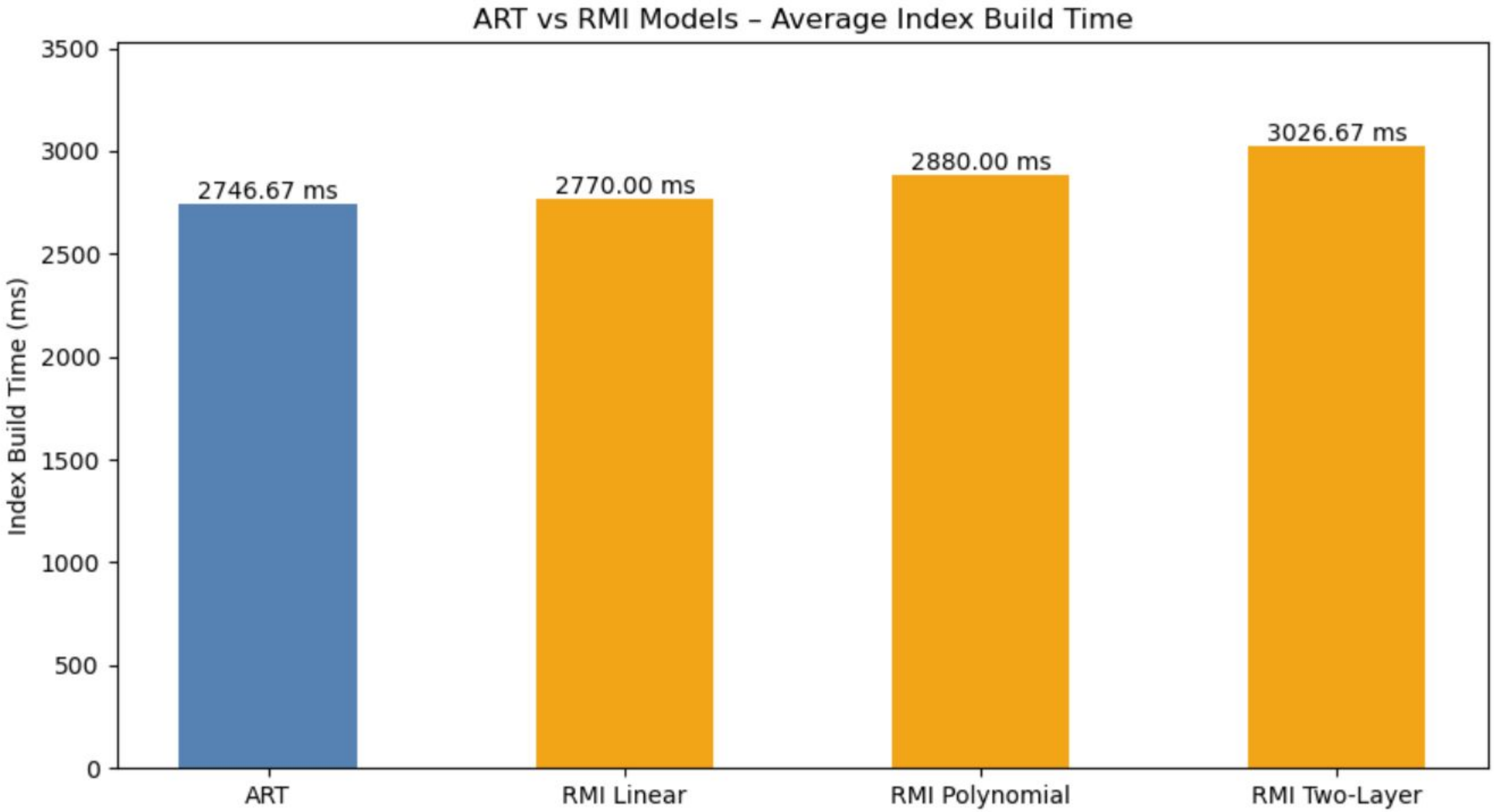


Results - Hit Rate



Results - Index Creation Time

USC Viterbi School of Engineering



Observations

- **Lightweight Index:** RMI index size is drastically smaller, roughly 1/3rd of ART, since RMIs store only lightweight model parameters instead of large pointer-heavy tree structures.
- **Execution time is comparable or better**: RMI models consistently outperform ART by 10–15 ms on average, providing faster point-lookup performance.
- **Index build time is marginally higher**: RMIs require model training during index construction, leading to longer creation times compared to ART.
- **Robust Index:** Hit rate is effectively 100%, indicating that the RMI predictions and error bounds are highly reliable in locating the correct key range.

Conclusions

USC Viterbi School of Engineering

Takeaways

- RMI indexes are practical inside DuckDB
- They offer strong performance on structured data
- They heavily reduce index memory footprint

Future work

- Adaptive segmentation
- Paging support for large outputs
- Deep Learning RMI models

Thank You! Any Questions?