Universität des Saarlandes

From Component to System: Evaluating Recursive Model Indexes and Index Scan Execution Strategies

Marcel Maltry

A dissertation submitted towards the degree
Doctor of Engineering (Dr.-Ing.)
of the Faculty of Mathematics and Computer Science
of Saarland University

Saarbrücken, 2024

Day of Colloquium
November 20, 2024

Dean of the Faculty
Prof. Dr. Roland Speicher

Examination Board
Chair
Prof. Dr. Jörg Hoffmann

Reviewers
Prof. Dr. Jens Dittrich
Prof. Dr. Sebastian Michel

Academic Assistant
Dr. Daniel Höller

Abstract

Indexes are crucial in database systems for ensuring efficient data retrieval and achieving peak query performance. Traditionally, indexes are general-purpose data structures, agnostic to data distributions. However, learned indexes, like the recursive model index (RMI), challenge this conventional approach by using machine-learned models that exploit data-specific patterns. This allows learned indexes to achieve both remarkable lookup performance and better space efficiency than traditional indexes.

Constructing an RMI requires precise adjustment of multiple hyperparameters to attain optimal performance. Therefore, we conduct a thorough analysis of the RMI's hyperparameters in the first part of this thesis, developing a practical guideline for its configuration. Compared to exhaustive enumeration, our guideline achieves competitive lookup performance while being more efficient, training at most two RMIs. The performance is also validated against state-of-the-art indexes, both learned and traditional.

We integrate RMIs into a database system with a compiling query engine in the second part of this thesis. We present a novel query engine that allows the query compiler to partially execute queries during compilation. Based on this architecture, we develop three strategies for accessing indexes in the context of an index scan. These strategies differ in which steps of the index scan are executed during compilation versus at runtime. We evaluate the three strategies based on query execution time, identifying the ideal query selectivities and workload characteristics for each. Despite their benefits, our experiments show that RMIs do not improve query performance of index scans compared to a simple baseline index, as index traversal time proved negligible compared to overall query execution time.

Zusammenfassung

Indexe sind in Datenbanksystemen von entscheidender Bedeutung, um effiziente Datenabrufe zu gewährleisten und optimale Abfrageleistung zu erzielen. Üblicherweise sind Indexe universelle Datenstrukturen, deren Leistung unabhängig von der Datenverteilung ist. Gelernte Indexe wie der Recursive-Model-Index (RMI) stellen diesen konventionellen Ansatz jedoch zunehmend infrage, indem sie mittels maschinellem Lernen datenspezifische Muster ausnutzen und so Indizes durch gelernte Modelle ersetzen. Dadurch erzielen gelernte Indexe bemerkenswerte Abfrageleistung und sind zudem speicherplatzsparender als herkömmliche Indexe.

Die Konstruktion eines RMI erfordert präzises Anpassen mehrerer Hyperparamter, um optimale Leistung zu erzielen. Daher führen wir im ersten Teil der Arbeit eine umfassende Analyse der Hyperparameter des RMI durch und entwickeln eine praxisorientierte Anleitung zum Konfigurieren von RMIs. Verglichen mit vollständigem Aufzählen aller Konfigurationen erzielt unser Ansatz konkurrenzfähige Abfrageleistung und ist gleichzeitig effizienter, da höchstens zwei RMIs trainiert werden müssen. Die Leistung wird zudem gegenüber hochmodernen Indexen, sowohl traditionellen als auch gelernten, validiert.

Im zweiten Teil der Arbeit integrieren wir RMIs in ein Datenbanksystem mit kompilierender Query-Engine. Wir präsentieren eine neuartige Query-Engine, die es dem Query-Compiler ermöglicht, Abfragen bereits während der Kompilierung teilweise auszuführen. Basierend auf dieser Architektur entwickeln wir drei Strategien für den Indexzugriff im Kontext eines Index-Scans. Diese Strategien unterscheiden sich darin, welche Schritte des Index-Scans während des Kompilierens und welche zur Laufzeit ausgeführt werden. Wir vergleichen die drei Strategien hinsichtlich Abfrageleistung und ermitteln für jede Strategie die optimale Abfrageselektivitäten und Workload-Eigenschaften. Trotz ihrer Vorteile zeigen unsere Experimente, dass RMIs die Abfrageleistung von Index-Scans im Vergleich zu einem einfachen Referenzindex nicht verbessern, da die Indexzugriffszeit im Verhältnis zur gesamten Abfrageausführungszeit vernachlässigbar ist.

Acknowledgements

First and foremost, I would like to express my sincerest gratitude to my advisor, Prof. Dr. Jens Dittrich, for granting me the opportunity to pursue this doctoral research. His invaluable guidance and insightful feedback have balanced academic freedom with academic support, shaping my journey and contributing to the success of my research endeavors. His distinctive approach to presenting research has left a lasting impact on me, and I am immensely thankful for his mentorship and influence.

I would also like to thank Prof. Dr. Sebastian Michel for his constructive feedback as the second reviewer of this thesis. His extensive expertise in our field of research has greatly contributed to the refinement of this work.

Further, I am grateful for my current and former colleagues and friends, with whom I have engaged in countless hours of fruitful discussions, spanning database-related topics and beyond, both at work and in our private lives. Your companionship and support over the years mean a great deal to me. I extend special thanks to Jannik, Joris, and Luca for their invaluable help in proofreading this thesis.

I am deeply thankful to my family for their continuous support. My heartfelt gratitude go to my wife, Elena, for her unwavering love and patience, which have been my greatest source of strength, and my son, Marlon, whose presence has enriched my life in ways I never imagined, constantly challenging me to question my worldview and priorities.

Additionally, I would like to acknowledge the use of OpenAI's ChatGPT (versions GPT-4o and GPT-3.5) to assist with improving and refining my writing and identifying grammatical errors.

Contents

Chapter 1

Introduction

1.1 Motivation

The efficient storage and fast retrieval of data are paramount requirements across various domains, including business operations, financial services, healthcare, e-commerce, and public administration. These operations rely heavily on database systems, which form the backbone of virtually all data management processes. For instance, e-commerce platforms use database systems to handle millions of transactions per day, ensuring real-time inventory updates and personalized customer experiences. In healthcare, database systems manage vast amounts of patient data, supporting everything from electronic health records to complex analytics for medical research. These examples illustrate the demanding requirements for database systems in terms of performance, reliability, and scalability. To fulfill these challenging requirements, database systems employ a wide range of techniques.

Among these techniques, indexes stand out as one of the most significant methods for fast data retrieval, and they are the central focus of this thesis. An index is a specialized data structure that enables access to specific data items without having to inspect the entire dataset. There are different types of queries, each requiring specific types of indexes to efficiently support these queries. The three most relevant query types, along with commonly used indexes that support them, are listed in the following.

Point Queries – Point queries typically retrieve a single data item based on a specific key, such as finding detailed information about a student with a particular matriculation number in a student database. Point queries are commonly processed using hash indexes, which apply

a hash function to the key to quickly obtain the memory location of the requested data item. Additionally, B-trees support point queries by traversing their tree structure based on the key until reaching a leaf node that contains the desired data item.

Range Queries – Range queries retrieve a set of data items with values falling within a specified range. For instance, a range query might seek information on all students born between the years 2000 and 2003. Range queries are efficiently processed using tree-structured indexes like B-trees or tries. Typically, to retrieve the desired data items, the query initiates with a point query on the first value in the range, leading to the first qualifying data item located in some leaf node. Continuing from there, the query scans the remainder of the leaf node and the subsequent leaf nodes until the data items no longer fall within the specified range of values.

Existence Queries – Existence queries determine whether a data item with a specific value is present in the database, such as a student with a particular matriculation number. Both hash indexes and B-trees support this type of query by performing a point query on the specified value. Another commonly used index for existence queries is the Bloom filter. This probabilistic data structure efficiently processes existence queries by using multiple hash functions to set and read bits in a bit array. Bloom filters may produce false positives, so in case of a positive result, the actual data must be checked to confirm the existence of the data item.

The aforementioned indexes have in common that they are general-purpose data structures that make no assumptions on the distribution or inherent properties of the data they index. In fact, most components in database systems are designed to be general-purpose to support a wide range of workloads and datasets, and to adapt to changes in these workloads and datasets over time. For instance, query optimizers, although they use statistics to make informed decisions, are designed to be general-purpose, computing efficient query execution plans for a wide variety of workloads and data distributions, rather than being specialized for any single workload and dataset.

However, the invention of *learned indexes* [37] has challenged the generality of both individual database components and database systems as a whole. Learned indexes are based on the observation that traditional indexes function as models of the underlying data. By exploiting patterns and distributions within the data, learned indexes can significantly improve lookup performance and space efficiency. These learned indexes leverage machine learning techniques to replace traditional indexes with learned models. The success of learned indexes sparked a new line of research into instance-optimized database systems, where machine learning is employed to optimize components for specific datasets and workloads. This research extends to

areas including but not limited to storage layouts (e.g., [69, 13]), cardinality estimation (e.g., [28, 66, 64]), approximate query processing (e.g., [42, 70]), and even query optimizers (e.g., [48, 50]).

This thesis is dedicated to investigating the first learned range index, known as the recursive model index (RMI). We aim to understand the sources of its improved performance and efficiency over traditional indexes. By analyzing the hyperparameters involved in configuring the RMI, we develop a practical guideline for its configuration. We conduct a performance comparison between the RMI, traditional indexes, and other first-generation learned indexes, which refer to the initial wave of learned indexes that emphasize lookup performance while having limited or no support for updates. Furthermore, we examine the performance impact of using RMIs in a system, particularly focusing on its effectiveness in performing index scans as part of range queries. Additionally, we develop three strategies for accessing indexes in the context of an index scan in a compiling database engine and conduct a comprehensive comparison of these strategies to determine the best use case for each. Through detailed analysis and experimentation, this thesis seeks to investigate the potential of learned indexes to enhance database efficiency.

1.2 Project Genesis and Research Questions

Building on the motivation outlined above, this thesis examines indexes and their use from two distinct perspectives: the *component view* and the *system view*. Chapter 2 takes on the component view by analyzing and evaluating the performance of a specific index, the RMI, in isolation. We shift towards the system view in Chapter 3, investigating the performance impact of different strategies for accessing indexes in an index scan within a database system. Both perspectives are crucial for a holistic understanding of index performance. The component view provides foundational insights into the capabilities and limitations of the RMI, while the system view demonstrates the practical implications and benefits of using indexes as part of index scans. In the following sections, we will delve into the genesis of both research projects that form the basis of Chapter 2 and 3 and outline their key research questions.

Component View – A Critical Analysis of Recursive Model Indexes

When the RMI was first introduced in December 2017 by Kraska et al. [36], it was met with both recognition and skepticism [52, 4]. The novelty and impressive experimental results brought much-needed innovation to a stagnant field of database research. However, there was skepticism due to imprecise experiment descriptions and the absence of an open-source implementation and accessible datasets. RMIs are based on the simple observation that

the position of a key in a sorted array can be computed using the cumulative distribution function (CDF) of the underlying data. RMIs approximate this CDF through a hierarchical model to predict the position of a given key in the sorted array. Despite the simplicity of this concept, configuring RMIs involves precisely adjusting several hyperparameters, including model types, number of layers, and layer sizes.

Driven by the desire to better understand RMIs and to reproduce the results from the paper, we initially investigated RMIs as part of a Bachelor's thesis. However, efficiently reimplementing RMIs based on the paper proved challenging due to a lack of detailed information. After intensive consultation with the authors, we managed to close some of the gaps and evaluate an initial prototype implementation. While we were able to reproduce some of the key results, our goal was to deepen our understanding of the intricacies of RMIs – their strengths and weaknesses – and, to a certain degree, demystify RMIs and address some misconceptions about them in the community.

Two years after the initial publication of the original paper, in 2019, colleagues of the author finally published a reference implementation of RMIs along with an automatic optimizer for configuring them [47]. Although this reference implementation differed from the original description in some aspects, it enabled us to further improve our implementation to a point where a comprehensive analysis was possible. Building on our initial exploration, we aimed to address the following research questions to deepen our understanding of RMIs.

1. Are the results of the original paper reproducible?
One of our primary goals is to validate whether the results reported by Kraska et al. [37] are reproducible. Given the difficulties in replicating the original experiments, we aim to assess the performance of our independent implementation of RMIs and compare them against the reported results.

2. How do the hyperparameters involved in configuring RMIs affect performance?
By closely examining the influence of various hyperparameters on the overall performance of RMIs, we aim to identify patterns that indicate the optimal configuration for different scenarios. Understanding these relationships is crucial for answering the next research question.

3. Is there a practical way for configuring RMIs beyond exhaustive enumeration?
The automatic optimizer of the reference implementation determines configurations through time-consuming exhaustive enumeration. This process, while thorough, is not practical for many real-world applications. Leveraging the insights gained from our hyperparameter

analysis, we aim to develop a more efficient and practical guideline for configuring RMIs. Our goal is to simplify the configuration process, making it feasible for a wider range of use cases.

4. How do RMIs compare to other indexes in terms of performance and efficiency? The performance benefits of RMIs cannot be fully evaluated without comparing them to other indexes. To assess the performance of RMIs, we select a variety of indexes, both traditional and learned, for comparison. This analysis will highlight the strengths and weaknesses of RMIs relative to other options. Our goal is to identify specific scenarios where the use of RMIs may be particularly advantageous.

5. How do RMIs and other learned indexes balance prediction and error correction? Since all first-generation learned indexes approximate the CDF, we aim to investigate how the different approaches to this approximation impact performance. Our analysis will not only focus on pure lookup time but will also decompose performance into the prediction of the position in the sorted array and the correction of imprecise predictions. By doing so, we seek to determine whether one approach clearly dominates the others in terms of overall efficiency.

In conclusion, this study adopts the component view to investigate the performance and efficiency of RMIs under ideal conditions. By isolating RMIs from a system and repeatedly querying them, we enable the CPU to optimize cache usage. Our focus on the internal workings and configurations of RMIs aims to provide valuable insights into their capabilities and performance characteristics. While limited to the component view, this study serves as a foundational step towards understanding the role of learned indexes in enhancing database efficiency. However, it remains uncertain how the observed performance numbers transfer to usage of RMIs in a real system.

System View – Index Access Strategies for Index Scans

Having thoroughly investigated RMIs in isolation, the natural next step was to evaluate them in a real system. We decided to integrate RMIs into mu*t*able [26], a database system designed for research and fast prototyping, currently under development in our group. At the project's inception, mu*t*able entirely lacked support for indexes, so we chose to focus on implementing an index scan operator to enhance range queries.

mu*t*able has a unique compiling query engine that generates WebAssembly code from an optimized query execution plan (QEP) and then executes this WebAssembly code in an embedded runtime. The embedded runtime isolates the execution from the host database system, preventing direct access to host data, including tables and indexes. To circumvent this

restriction, tables are rewired [56] to the WebAssembly program's memory region, allowing for direct access. However, this approach requires reimplementing the entire table access logic in WebAssembly. While rewiring and reimplementing access logic would be possible for indexes, it would demand significant development effort, not only once but whenever a new type of index is added. An alternative and *de facto* standard approach to access host data from within the embedded runtime involves using host calls. Although this method is flexible and easy to implement, host calls typically induce overhead due to context switching and marshalling of function parameters and return values, potentially deteriorating performance.

To circumvent the restrictions of executing queries in an isolated environment, we developed a novel query engine architecture that allows partial execution of QEPs during compilation using an interpreter. This architecture enables certain operator, including parts of an index scan, to be performed at compile time rather than at query runtime. Building on this foundation, we examined the individual steps of an index scan and observed that the index is primarily used to determine tuple IDs, which are the only required information within the embedded runtime. Based on this observation, we developed three index access strategies as part of an index scan in a compiling database engine: (1) compiled index access strategy, (2) interpreted index access strategy, and (3) hybrid index access strategy. While these strategies were designed with mu*t*able's execution model in mind, they are not limited to use within an isolated execution environment and can also be applied in an integrated setting.

Initially centered around the application of RMIs in a real system, this project evolved into a broader investigation of a novel query engine design and different strategies for accessing indexes in the context of index scans. Despite this broader focus, RMIs remain a critical component of our study. By comparing the performance impact of these strategies across two indexes, including RMIs, we aim to highlight the specific advantages and limitations of each strategy. This comprehensive analysis is guided by the following research questions.

1. When to use which strategy to achieve the best query execution performance?
By executing queries with range filter predicates of varying selectivity on different attribute types, we aim to investigate if one of the strategies clearly dominates the others in terms of query execution time. If no single strategy consistently outperforms the others, we will identify the best use cases for each strategy. By using simple queries, we aim to minimize the influence of extraneous factors, ensuring that our evaluation focuses solely on the performance of each strategy.

2. How to properly configure each index scan strategy?
Each index access strategy includes one or more configuration parameters. By evaluating a

wide range of selectivities, we will investigate the performance impact of these parameters to identify the optimal settings for each strategy.

3. Can certain strategies benefit from caching compiled plans?
In two strategies, the generated code does not contain the filter predicate because the index is accessed during query compilation. This prompts an important question: can cached plans from similar queries be reused in these strategies? If so, it would eliminate the need for recompilation, potentially reducing the compilation time of future similar queries and even improving query execution time overall.

4. Can RMIs enhance the performance of index scans?
The index access strategies can be implemented with any range index. Our investigation aims to determine whether using an RMI can improve the performance compared to a simple baseline index.

In conclusion, this project adopts a system view to explore the integration of RMIs into the mu*t*able database system, focusing on enhancing range queries through index access strategies. By implementing these strategies and conducting rigorous performance evaluations, our aim is to uncover insights into the effectiveness and optimization of these strategies within real-world systems. While our findings offer valuable insights, further research is essential to consolidate conclusions regarding the utilization of learned indexes.

1.3 Contributions

This section presents a detailed overview of the key contributions made through the projects covered in the subsequent chapters of this thesis. Each project addresses the aforementioned research objectives and challenges.

Component View – A Critical Analysis of Recursive Model Indexes

Most of Chapter 2 has been previously published in:

> Marcel Maltry and Jens Dittrich. "A Critical Analysis of Recursive Model Indexes." In: *Proc. VLDB Endow.* 15.5 (2022), pp. 1079–1091

We made the following contributions through this project.

1. Learned Index Review – We offer an overview of the extensive body of work on learned indexes, contrasting RMIs with other first-generation learned indexes, and distinguish our

study from previous experimental research focused on comparing and understanding the performance of learned indexes.

2. Training Optimization - We provide a comprehensive examination of RMIs by detailing their core concepts, elucidating the mechanisms of lookups, and identifying the key hyperparameters for their effective configuration. Through thorough analysis of the training process, we can improve training time by leveraging the monotonicity of the models used in RMIs.

3. Hyperparameter Analysis – We define our methodology and experimental setup in detail, including the tested hyperparameter configurations, datasets, workloads, and other indexes used as performance baselines. We conduct the first inventor-independent extensive hyperparameter analysis and determine the impact of each hyperparameter on prediction accuracy, lookup performance, and build time.

4. Configuration Guideline – Building on our hyperparameter analysis, we develop a practical guideline for configuring RMIs. This guideline, which accepts a size budget for the index, offers a simpler alternative to previous approaches that relied on extensive enumeration. Although our method does not guarantee the best configuration in terms of lookup performance, it significantly reduces training time by limiting the process to building at most two instances of an RMI . Experimental results demonstrate that our guideline achieves performance similar to the optimal configuration on datasets without extreme outliers.

5. Index Comparison – We compare our implementation of RMIs against the previously released reference implementation of RMIs, as well as all publicly available first-generation learned indexes, and a selection of traditional indexes, in terms of lookup time and build time. Our implementation is configured using our guideline, while the reference implementation uses exhaustive enumeration. We show that despite the simple configuration process, on datasets without extreme outliers, our guideline is *on par* or even outperforms the reference implementation. We also demonstrate the scalability of the indexes across datasets of varying sizes. Additionally, we conduct a breakdown of lookup time into the time taken to traverse the index and the time to search for the concrete entry, either in the leaf nodes of the index or a flat array, analyzing how different indexes prioritize optimizing these aspects.

6. Open-Source Implementation – Our complete code consisting of an extensible implementation of RMIs and experiments, including scripts for reproducing our measurements and plots, is publicly available on GitHub [43]. The code is licensed under Apache-2.0. Furthermore, the experiments were deemed reproducible by the reproducibility committee of VLDB 2022.

System View – Index Access Strategies for Index Scans

We made the following contributions through this project.

1. Query Processing Review – We revisit the steps involved in processing a query in a modern database system, explaining query execution by both interpretation and compilation with a simple example. This example is further utilized to illustrate the index scan operation, breaking down the operator into its multiple constituent steps.

2. Query Compilation With Partial Execution – We present a novel compiling query engine architecture that allows the query compiler to partially execute QEPs during compilation using an interpreter. Runtime observations, such as intermediate results, are leveraged to enhance the generated code with additional information, enabling further optimizations by the execution engine.

3. Index Access Strategies – Based on this architecture, we develop three distinct strategies for accessing indexes in the context of an index scan. These strategies differ in terms of which of the steps involved in an index scan are carried out during query compilation and which are executed at query runtime. We identify different methods of implementing each strategy, varying in how intermediate results are materialized and how the index is accessed based on the execution environment. Additionally, we discuss other potential strategies and provide arguments for why we consider them to be inferior.

4. Strategy Comparison – We conduct a comprehensive experimental study comparing the performance of multiple variants of the three strategies, aiming to understand when each is most effective. We consider both a simple sorted array and a recursive model index as basis of the index scan to determine the performance impact of using a learned index. Additionally, we examine whether caching compiled plans enhances the strategies by allowing previously compiled plans to be reused for similar queries.

5. Open-Source Implementation – We illustrate the implementation of the three index access strategies in mu*t*able, a database system currently under development in our group. We describe mu*t*able's query engine, which translates queries to WebAssembly and executes them in an embedded runtime isolated from the rest of the system. Additionally, we explain the implications of this system design on the implementation of the three index access strategies. The complete code for the strategy implementations and experiments is publicly available on GitHub [27] as part of mu*t*able and licensed under AGPL-3.0.

Chapter 2

A Critical Analysis of Recursive Model Indexes

2.1 Introduction

Machine learning and artificial intelligence are taking the world by storm. Research areas that were believed to have been researched to completion have been revisited with exciting new results, showing that considerable improvements are still possible *if* we factor in wisdom from the machine learning world. Notable examples include natural language processing and computer vision, which were completely revolutionized in the past decade by variants of deep learning. In the database world, we witnessed a surge of similar reexploration endeavors in the past five years. Recent notable examples of work in this space include cardinality estimation [28, 66], auto-tuning [54, 2], and indexing [37, 18, 12, 16, 34]. We believe that indexing is the most surprising result of these three areas because both cardinality estimation and auto-tuning are optimization problems and thus naturally align with machine learning. The connection to indexing becomes evident when we examine a special case of indexing.

Problem Statement – Given a sorted, densely packed array A of keys and a query Q asking for a particular key x_i that may or may not exist in that array, return the array index i of that key x_i.

In other words, we are looking for a function that assigns to each key its position in the sorted array. Traditionally, this function is implemented by a suitable algorithm like binary search or a data structure like a B-tree. In contrast, Kraska et al. [37] observe that this function can

be learned through regression, effectively making the indexing problem a machine learning task. Based on this observation, Kraska et al. [37] present the recursive model index (RMI) as the first *learned index* with remarkable results in terms of lookup performance. We wanted to understand the performance benefits of RMIs early on and therefore tried to reproduce the results. However, we quickly encountered several issues.

Hyperparameter configuration – Configuring RMIs involves setting several hyperparameters. Unfortunately, the exact configurations with which the remarkable results were obtained were not reported and, in some cases, even described misleadingly. The use of neural networks is mentioned frequently throughout the experimental evaluation of the original paper. However, the low model evaluation times reported in Fig. 4 strongly suggest that none of the best-performing configurations actually used neural networks. In personal communication with the first author in August 2019, we learned that linear models should be preferred over neural networks in most cases. In our experience, there is still a misconception in the community today that RMIs internally use neural networks. Subsequent studies [33, 49] involving inventors of the RMI investigated the performance benefits of learned indexes over traditional indexes. However, hyperparameter configurations for the reported results were obtained by a time-consuming enumeration process [47]. As a result, similar to the original paper [37], the studies neither show how the choice of hyperparameters affects performance nor do they give advice for configuring RMIs in practice besides enumeration.

Closed source – The source code of the original paper was never made available. A so-called reference implementation [49], which differs from the descriptions in the original paper (see Section 2.3.2), was published in December 2019, two years after the preprint [36].

Goals – We pursue the following objectives with this paper:

1. Conduct the first inventor-independent detailed analysis of RMIs to understand the impact of each hyperparameter on prediction accuracy, lookup time, and build time.
2. Develop a clear and simple guideline for database architects on how to configure RMIs with good lookup performance.
3. Provide a clean and easily extensible implementation of RMIs.

Contributions – We make the following contributions to achieve these goals:

1. **Learned Indexes**: We revisit in detail recursive model indexes [37] and explain how they are trained and what hyperparameters to consider (Section 2.2). We provide a detailed

overview on the design dimensions of learned indexes and the already large body of work in that space (Section 2.3).

2. **Hyperparameter Analysis**: We present our experimental setup (Section 2.4) and conduct a set of extensive experiments to analyze the impact of each hyperparameter on predictive accuracy and error interval size (Section 2.5), lookup performance (Section 2.6), and build time (Section 2.7).

3. **Configuration Guideline**: Based on our findings, we develop a simple guideline to configure RMIs in practice (Section 2.8).

4. **Comparison With Other Indexes**: We compare the RMIs resulting from our guideline in terms of lookup time and build time with a number of learned indexes like ALEX [12], PGM-index [16], RadixSpline [34], and the reference implementation of RMIs [47], as well as state-of-the-art traditional indexes like B-tree [5], ART [39], and Hist-Tree [10] (Section 2.9).

2.2 Recursive Model Indexes

In this section, we review recursive model indexes, covering the lookup process, training methodology, and relevant hyperparameters.

2.2.1 Core Idea

RMIs are based on the observation that the position of a key in a sorted array can be computed using the cumulative distribution function (CDF) of the data. Let D be a dataset consisting of $n = |D|$ keys. Further, let X be a random variable that takes each key's value with equal probability, and let F_X be the CDF of X. Then, the position i of each key $x_i \in D$ in the sorted array is computed as:

$$i = F_X(x_i) \cdot n = P(X \leq x_i) \cdot n \tag{2.1}$$

Note that in the context of learned indexes, the term CDF is frequently used synonymously for a mapping from key to position in the sorted array instead of its statistical definition as a mapping from key to the probability that a random variable will take a value less than or equal to that key. In the following, we adopt the former interpretation.

The core idea of an RMI is to approximate the CDF of a dataset using a hierarchical, multi-layer model. Consider Figure 2.1 for an example of a three-layer RMI. Each model in an RMI approximates a segment of the CDF; all models of a layer together approximate the entire CDF.

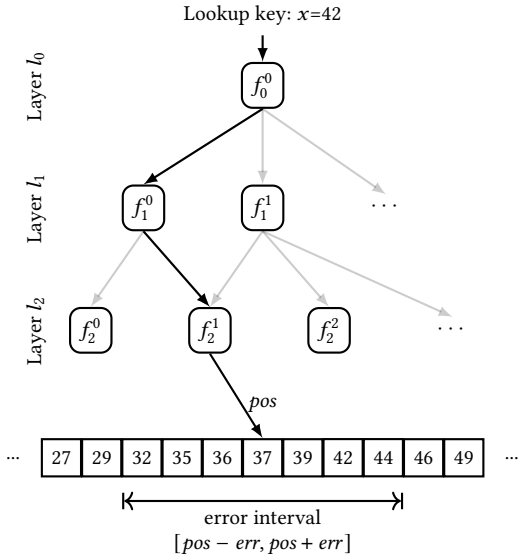


Figure 2.1: Illustration of a three-layer RMI evaluated on key 42, producing an estimated position *pos* (prediction). The estimated position *pos* is used to locate the key within the sorted array by searching the error interval around the predicted position (error correction).

An RMI is best represented as a directed acyclic graph. Unlike a tree, a node (or model) in an RMI may have multiple direct predecessors. We denote the i-th layer of a k-layered RMI by l_i where i ranges from 0 to $k-1$, and refer to the j-th model of the i-th layer by f_i^j. The first layer l_0 of an RMI always consists of a single root model f_0^0. Each subsequent layer may consist of an arbitrary number of models. The number of models of a layer l_i is denoted by $|l_i|$ and is called the size of the layer.

2.2.2 Index Lookup

A lookup is performed in two steps:

1. **Prediction**: We evaluate the RMI on a given key yielding a position estimate.
2. **Error Correction**: We search the key in the area around the estimated position in the sorted array to compensate for estimation errors.

We discuss both steps in more detail below.

Prediction – Consider again Figure 2.1, which shows an index lookup for key 42. We start by evaluating the root model f_0^0 on key 42, yielding a position estimate. Based on this estimate,

model f_1^0 in the next layer l_1 is chosen for evaluation. This iterative process is continued until the position estimate *pos* of the last layer is obtained.

Definition (Prediction) Let R be a k-layer RMI trained on dataset D consisting of $n = |D|$ keys. Denote the value p restricted to the interval $[a, b]$ by

$$[\![p]\!]_a^b := \max\big(a, \min(p, b)\big). \tag{2.2}$$

The predicted position for key x of layer l_i is recursively defined as

$$f_i(x) = \begin{cases} f_0^0(x) & i = 0 \\ f_i^{\left\lfloor [\![|l_i| \cdot f_{i-1}(x)/n]\!]_0^{|l_i|-1} \right\rfloor}(x) & 0 < i < k \end{cases} \tag{2.3}$$

Intuitively, to determine the model in layer l_i that is evaluated on key x, the estimate $f_{i-1}(x)$ of the previous layer is scaled to the size of the current layer. Note that $f_{i-1}(x)$ might be less than 0 or greater than $n - 1$. Thus, the result is restricted to $[0, |l_i| - 1]$ to evaluate to a valid model index. The predicted position for key x of RMI R is the output of layer l_{k-1}:

$$R(x) = f_{k-1}(x). \tag{2.4}$$

Error correction – Based on the estimate $R(x)$ obtained by evaluating the RMI, the sorted array is searched for the key. In Figure 2.1, the position estimate for key 42 points to key 37 in the sorted array. Since 37 is smaller than 42, we have to search to the right of 37 to find 42. To facilitate the search, an RMI may store error bounds that limit the size of the interval that has to be searched. The RMI guarantees that if a key is present, then it can be found within the provided error bounds. A simple way to achieve this is by storing the maximum absolute error *err* of the RMI. The left and right search bounds, i.e., the error interval, are set to $[R(x) - err, R(x) + err]$. If the key exists, it must be within these bounds. We search the interval for key x using an appropriate algorithm such as binary search.

2.2.3 Training Algorithm

The goal of the training process is to minimize the prediction error. The training algorithm is shown in Algorithm 1. Its core idea is to perform top-down, layer-wise bulk loading. We start by assigning all keys to the root model (Line 4). Then, the root model is trained on those keys (Line 7). Afterward, the keys are assigned to the next-layer models based on the root

model's estimates (Lines 9 to 11). We proceed by training the models of the next layer on the keys that were assigned to them. This process is repeated for each layer until the last layer has been trained. Finally, if desired, error bounds can be computed for the trained RMI (after Line 11).

Algorithm 1 RMI Training Algorithm

Input: Dataset D, number of layers k, array of layer sizes l
Output: RMI R

```
1:  procedure BuildRMI(D, k, l)
2:      R := Array2D()                          ▷Initialize dynamic array to store models.
3:      keys := Array2D()                       ▷Initialize dynamic array to store each model's keys.
4:      keys[0, 0] := D                         ▷Assign all keys to the root model.
5:      for i ← 0 to k − 1 do
6:          for j ← 0 to l[i] − 1 do
7:              R[i, j] := TrainModel(keys[i, j])        ▷Train model j of layer i.
8:              if i < k − 1 then                ▷Check whether current layer is not last layer.
9:                  for all x in keys[i, j] do
10:                     p := GetModelIndex(x, R[i, j], l[i + 1], |D|)
11:                     keys[i + 1, p].add(x)    ▷Assign key x to next-layer model p.
12:                 end for
13:             end if
14:         end for
15:     end for
16:     return R
17: end procedure

18: function GetModelIndex(x, f, q, n)
19:     return ⌊[[q · f(x)/n]]_0^{q−1}⌋          ▷Compute model index according to Equation (2.3).
20: end function
```

2.2.4 Hyperparameters

RMIs offer a high degree of flexibility in configuration and tuning. In the following, we briefly describe each hyperparameter. We provide a set of possible configurations for each parameter in Section 2.4.2 when describing the experimental setup.

Model Types – The choice of model types significantly influences the predictive quality of RMIs. Simple models like linear regression are compact and efficient in training and evaluation, whereas complex models such as neural networks may provide higher accuracy at the cost of slower training and evaluation speeds.

Layer Count – The number of layers k determines the depth of an RMI. Deeper RMIs may distribute keys more evenly across the last-layer models, but they are larger and take longer to train and evaluate.

Layer Sizes – The size of a layer defines the number of models within that layer. A larger number of models leads to more accurate predictions, since the segments that the models have to cover are smaller.

Error Bounds – Error bounds facilitate the error correction by limiting the size of the interval that has to be searched. Error bounds can be chosen on different granularities or be omitted altogether.

Search Algorithm – Depending on the error bounds, various search algorithms can be employed for error correction. Examples include binary, linear, or exponential search.

2.3 Related Work

The introduction of learned indexes by Kraska et al. [37] caused both excitement and criticism within the database community. Early criticism mainly focused on the lack of efficient updates, the relatively weak baselines, and the absence of an open-source implementation [52, 4]. Later, Crotty [10] argued that the performance advantages of learned indexes stem primarily from implicit assumptions on the data, such as sortedness and immutability. Subsequently published learned indexes addressed some of these weaknesses [18, 12, 16]. Nevertheless, the RMI remains one of the fastest indexes in experimental evaluations [33, 49, 34, 10].

2.3.1 Learned Indexes

Existing learned indexes commonly approximate the CDF. These indexes most notably differ in the type of model they use to approximate the CDF, whether they are trained bottom-up or top-down, and whether they support updates.

FITing-tree – FITing-tree [18] models the CDF using piecewise linear approximation (PLA). During training, a dataset is first divided into variable-sized segments by a greedy algorithm in a single pass over the data. These segments are created so that their linear approximation satisfies a user-defined error bound. Subsequently, segments are indexed by bulk loading them into a B-tree. A lookup consists of traversing the B-tree to find the segment that contains the key, computing an estimated position based on the linear approximation of the segment, and searching for the key within the error bounds around the estimated position. FITing-tree

supports inserts, which can either be performed in-place by shifting existing keys within the segment or using a buffering strategy. In the buffering strategy, each segment has a buffer that is merged with the other keys in the segment whenever the buffer is full. Unfortunately, at the time of writing, no open-source implementation of FITing-tree was available, which kept us from including it in our experiments.

ALEX – ALEX [12] uses a variable-depth tree structure to approximate the CDF with linear models. Internal nodes are linear models that, given a key, determine the appropriate child node. Leaf nodes store the data, whose distribution is also approximated using a linear model. During a lookup, the tree is traversed until a leaf node is reached. Then, a position is predicted using the leaf's linear model. Finally, the key is searched using exponential search. Similar to RMI, ALEX is trained top-down. However, ALEX has a dynamic structure that is controlled by a cost model, which decides how to split nodes. ALEX supports inserts by either splitting or expanding full nodes.

PGM-index – PGM-index [16] also approximates the CDF using PLA. Similar to FITing-tree, PGM-index starts by computing segments that adhere to a specified error bound. However, unlike FITing-Tree, PGM-index creates a PLA-model that is optimal in the number of segments. Each segment is represented by the smallest key in that segment and a linear function that approximates the segment. Afterward, this process is continued recursively bottom-up by again creating a PLA-model on the smallest keys of each segment. The recursion terminates when only a single segment remains. So, unlike ALEX, each path from the root model to a segment is of equal length. A lookup is an iterative process where on each level of the PGM-index, a linear model predicts the next-layer segment containing the key and the correct segment is searched within the error bounds around the prediction using binary search. This process continues for the next-layer segment until the sorted array of keys is reached. Ferragina and Vinciguerra [16] also introduce variants of PGM-index that support updates (dynamic PGM-index) and compression at the segment level (compressed PGM-index). The size of PGM-index depends on the number of segments required to satisfy the used-defined error bound.

RadixSpline – In contrast to the aforementioned learned indexes, RadixSpline [34] approximates the CDF using a linear spline. The linear spline is fitted in a single pass over the data and to satisfy a user-defined error bound. The resulting spline points are then inserted into a radix table, which maps keys to the smallest spline point sharing the same prefix. The size of the radix table is determined by a user-defined prefix length. A lookup consists of finding the spline points surrounding the lookup key using the radix table, performing linear interpolation

between the spline points to obtain an estimated position, and applying binary search in the error interval around the estimated position to locate the key. Similar to RMI, RadixSpline has a fixed number of layers and does not support updates.

2.3.2 Experiments and Analysis

Marcus, Zhang, and Kraska [47] released an open-source reference implementation of RMIs along with an automatic optimizer in December 2019. This reference implementation deviates from the original description [37] in several aspects. For instance, model types like B-tree nodes and neural networks are missing, and error bounds are determined on a different granularity. Given a dataset, the optimizer uses exhaustive enumeration to determine a set of Pareto-optimal (in terms of lookup time and index size) two-layer RMI configurations. These configurations consist of first-layer and second-layer model types, and a second-layer size. Instead of blindly performing this costly enumeration, our work aims to understand the impact of each hyperparameter and to develop a simple guideline. Further, in addition to model types and layer sizes, we also consider error bounds and search algorithms when configuring RMIs.

Kipf et al. [33] introduced the Search On Sorted Data (SOSD) benchmark, a benchmarking framework for learned indexes. Besides providing a variety of index implementations, they supply four real-world datasets. In their preliminary analysis, the authors conclude that RMI and RadixSpline are able to outperform traditional indexes including ART [39], FAST [31], and B-trees, despite being significantly smaller in size. The authors also state that the lack of efficient updates, lengthy building times, and the necessity for hyperparameter tuning are notable issues with learned indexes.

As a follow-up, Marcus et al. [49] conducted a more detailed experimental analysis of learned indexes based on the framework and datasets from SOSD [33]. The authors performed a series of experiments to explain the superior performance of learned indexes and conclude that a combination of fewer cache misses, branch misses, and instructions largely accounts for the improved performance compared to traditional indexes. Further, the authors show that learned indexes are Pareto optimal in terms of size and lookup performance across datasets and key sizes.

Both aforementioned studies [33, 49] involve inventors of the RMI and aim to explain the performance of learned indexes in general. Since the evaluated RMI configurations were obtained using the optimizer [47], the studies neither show the impact of incorrectly configuring an RMI, nor do the studies provide advice on how to configure RMIs outside of using the optimizer. In

contrast, to the best of our knowledge, we conduct the first independent and holistic analysis of RMIs that directly compares configurations and aims to explain their performance.

Ferragina, Lillo, and Vinciguerra [15] take a theoretical approach to understanding the benefits of learned indexes, particularly those based on PLA. The authors demonstrate that for various distributions, PGM-index [16], while achieving the same query time complexity as B-trees, offers improved space complexity. To support their theoretical results, the authors conduct several experiments both on synthetic and real-world datasets. The theoretical results provide a solid foundation for further research. However, since RMIs are not limited to PLA and do not aim to construct the optimal number of segments, the results cannot be transferred to RMIs.

2.4 Experimental Setup

In this section, we introduce the implementation, hyperparameters, datasets, and workload used in our experiments and baselines considered for comparison. All experiments are conducted on a Linux machine with an Intel® Xeon® E5-2620 v4 CPU (2.10 GHz, 512 KiB L1, 4 MiB L2, 20 MiB L3) and 4x8 GiB DDR4 RAM. Our code is compiled with `clang-12.0.1` at optimization level `-O2` and executed single-threaded.

2.4.1 Implementation

Our implementation of RMIs is written in C++. RMI classes have a fixed number of layers, and model types are passed as template arguments. This implies that all models in a layer are of the same type, consistent with the reference implementation but differing from the original paper [37], which allows individual models to be replaced by B-tree nodes if predication are insufficiently accurate. Training algorithms of the model types are adapted from the reference implementation [47].

When assigning keys to the next-layer models, the reference implementation always copies keys to a new array. We optimized the training process based on the observation that the models considered here are monotonic and will never create overlapping segments. Thus, when assigning keys to next-layer models, we simply store iterators on the sorted array of the first and last key of each segment. We then train the next-layer models by passing them the respective iterators and thereby avoid copying the keys.

Further, instead of training all models on a mapping from key to position in the sorted array, we train inner layers on a mapping from key to next-layer model index. The index is obtained by scaling the position to the size of the next layer, akin to Equation (2.3). In other words, we

Table 2.1: Evaluated model types

Abbreviation	Method	Formula
LR	Linear Regression	$f(x) = ax + b$
LS	Linear Spline	$f(x) = ax + b$
CS	Cubic Spline	$f(x) = ax^3 + bx^2 + cx + d$
RX	Radix	$f(x) = (x \ll a) \gg b$

train inner layers directly on a targeted equal-width segmentation. This approach eliminates a multiplication and division during lookup that would otherwise be required to compute the model index from the position estimate.

2.4.2 Hyperparameters

In the following, we provide a list of hyperparameter configurations evaluated in our experiments and briefly compare them to those considered by the optimizer of the reference implementation [47].

Model Types – Table 2.1 lists the model types considered in our evaluation. Linear regression (LR) is a linear model that minimizes the mean squared error. Linear spline (LS) and cubic spline (CS) fit a linear and cubic spline segment through the leftmost and rightmost data points of a segment, respectively. Radix (RX) eliminates the common prefix and maps keys to their most significant bits. Models most notably differ in three aspects:

1. **Built Time**: LS, CS, and RX are fast to build from the leftmost and rightmost key. LR, being a regression method, is built on all keys.
2. **Evaluation Time**: RX is the fastest to evaluate with just two bit shifts. LR and LS are equally fast to evaluate, while CS is the slowest.
3. **Predictive Quality**: LS and CS are spline techniques, their predictive quality is based on how representative the leftmost and rightmost keys are. LR minimizes the error across all keys. RX, being radix-based, is solely used for segmentation.

In addition to the four models listed, the optimizer [47] considers radix tables, histograms, and a specialized variant of linear regression (see Section 2.9.1) for the first layer, and cubic splines for the second layer. We decided to evaluate a smaller set of model types to analyze the impact

Table 2.2: Evaluated error bounds

Abbreviation	Method	Granularity
LInd	Local Individual [37]	Maximum +/- error per model
LAbs	Local Absolute [47]	Maximum absolute error per model
GInd	Global Individual	Maximum +/- error per RMI
GAbs	Global Absolute	Maximum absolute error per RMI
NB	No Bounds [37]	-

of model types in general. Since the optimizer always recommends LR for the second layer, we focus solely on LR and LS for the second layer.

Layer Count – Like the optimizer [47], we only consider two-layer RMIs. It was previously reported that in most cases, two layers are sufficient to accurately approximate a CDF [47, 49], which we verified for the considered datasets in preliminary experiments. We plan to explore multi-layer RMIs as part of future work.

Layer Size – We cover the same wide range of second-layer sizes between 2^6 and 2^{25} in power of two steps like the optimizer [47].

Error Bounds – We consider five different variants of error bounds listed in Table 2.2, which differ in the granularity of the stored bounds in two respects:

1. **Local vs. Global**: Error bounds may be computed either for each last-layer model (local) or for the entire RMI (global). Global bounds, while more memory efficient, are susceptible to outliers because the single largest error determines the error interval size of all lookups. Local bounds are more robust against outliers as an outlier only affects the respective model.

2. **Absolute vs. Individual**: We can either store the maximum absolute error (absolute) or both the maximum positive and negative error individually (individual). While the former is again more space-efficient, the latter allows for tighter bounds, especially if a model either overestimates or underestimates the actual position.

Additionally, we may choose not to store any bounds (NB). Both local individual (LInd) and NB were suggested by Kraska et al. [37]. The reference implementation supports local absolute bounds (LAbs) and NB, but the optimizer [47] consistently recommends LAbs.

Table 2.3: Evaluated search algorithms

Abbreviation	Method
Bin	Binary Search
MBin	Model-biased Binary Search [37]
MLin	Model-biased Linear Search
MExp	Model-biased Exponential Search [37]

Search Algorithm – The evaluated search algorithms are listed in Table 2.3. We generally distinguish between two types of search algorithms:

1. **Standard Search Algorithms**: These search algorithms do not use the predicted position from the RMI. They only consider the error bounds around the predicted position.

2. **Model-Biased Search Algorithms**: These search algorithms use the predicted position from the RMI as the starting point of the search [37].

Binary search is an example of a standard search algorithm. We search the key in the interval between the two error bounds and ignore the position estimate.

However, binary search can be adapted to become model-biased. Instead of choosing the middle element of the interval as a first comparison point, we pick the estimated position. Similarly, linear search and exponential search can be adjusted to become model-biased. Instead of searching the interval from left to right, we start the search from the estimated position and search to the left or right, depending on whether the prediction is an overestimation or an underestimation. The search terminates once it is certain that the key cannot be found further in that direction. Initially, we also considered standard linear search and exponential search for our experiments, but both consistently performed worse than their model-biased counterparts.

It is worth noting that not all combinations of error bounds and search algorithms make sense. For instance, in the case of absolute error bounds, model-biased binary search and standard binary search are essentially the same as the estimate is the center of the interval anyway. Further, model-biased linear and exponential search do not make use of bounds.

Previous studies only compared binary [37, 33, 49], linear, and interpolation search [49]. Model-biased variants of linear and exponential search have not been studied in the context of RMIs thus far.

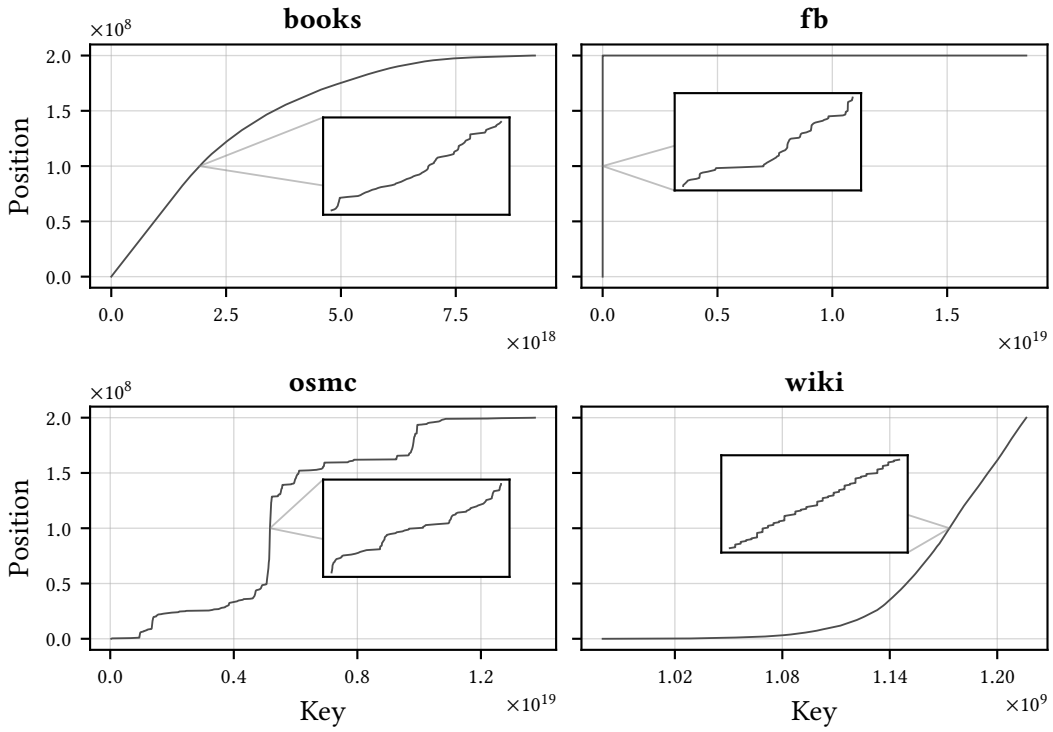


Figure 2.2: CDFs of the four real-world datasets from SOSD [33], each consisting of 200M 64-bit unsigned integer keys. Zoom-ins depict 100 consecutive keys.

2.4.3 Datasets

Learned indexes are known to adapt well to artificial data sampled from statistical distributions [49]. Therefore, we use the four real-world datasets from the SOSD benchmark [33]. Each dataset consists of 200M 64-bit unsigned integer keys. The CDFs of the four datasets are depicted in Figure 2.2; zoom-ins show 100 consecutive keys to indicate the amount of noise.

- **books**: Keys represent the popularity of books on Amazon.
- **fb**: Keys represent Facebook user IDs. This dataset contains a small number of extreme outliers, which are several orders of magnitude larger than the rest of the keys at the upper end of the key space. These outliers were not plotted in previous studies [33, 49].
- **osmc**: Keys represent cell IDs on OpenStreetMap. This dataset exhibits clusters that are artifacts of projecting two-dimensional data into one-dimensional space [49].
- **wiki**: Keys are Wikipedia edit timestamps. This dataset uniquely contains duplicates.

Table 2.4: Overview of the considered baselines

Method	Type	Hyperparameters	Source
RMI (ref) [37]	Learned	Model types, layer size	[45]
ALEX [12]		Sparsity	[11]
PGM-index [16]		Maximum error	[63]
RadixSpline [34]		Radix width, maximum error	[32]
B-tree [5]	Tree	Sparsity	[6]
Hist-Tree [10]		Number of bins, maximum error	[59]
ART [39]	Trie	Sparsity	[46]
Binary search	Search	-	[9]

2.4.4 Workload

For the lookup performance, we consider lower bound queries where the index returns an iterator pointing to the smallest element in the sorted array that is equal to or greater than the queried key. The sorted array is maintained in memory. We perform 20M lookups per run, where the keys are sampled from the sorted array uniformly at random with a fixed seed. Reported execution times are the average execution time of the median of three separate runs.

2.4.5 Baselines

In Section 2.9, we compare our RMI implementation against several baselines listed in Table 2.4, for which we use the referenced open-source implementations. Due to our focus on ranking the performance of RMIs, we consider all publicly available learned indexes at the time of writing, but only some representatives of traditional indexes.

Learned Indexes – ALEX [12], PGM-index [16], and RadixSpline [34] are learned indexes discussed in Section 2.3.1. The index size of PGM-index and RadixSpline is varied based on the maximum error parameter. Additionally, RadixSpline provides a parameter to adjust the size of the radix table that is used to index the spline points. Since we do not consider update performance here, we use the standard variant of PGM-index, which does not support updates. ALEX does not provide any parameters itself, so we vary its size by adjusting the number indexed keys (sparsity) by inserting only every k-th key. In addition, we also consider the reference implementation of RMIs (RMI (ref)) [47], configured using its integrated optimizer.

Traditional Indexes – B-tree [5] and ART [39] are traditional in-memory index structures. We vary the size of both B-tree and ART by adjusting the number of keys that are inserted. Therefore, we use an implementation of ART that supports lower bound queries from SOSD [33]. The recently published Hist-Tree [10] is a tree-structured index. Each inner node in a Hist-Tree is a histogram that partitions the data into equal-width bins. Like learned indexes, Hist-Tree exploits that the data is sorted. Hist-Tree provides two tuning parameters: the number of bins determines the size of inner nodes and the maximum error defines a threshold for the size of a terminal node. We use an implementation of a Compact Hist-Tree that does not support updates in favor of lookup performance [59].

Binary Search – We also consider standard binary search over the sorted array as provided by `std::lower_bound` [9] without any additional indexing.

2.5 Predictive Accuracy Analysis

In this section, we analyze the impact of hyperparameters on the predictive accuracy of RMIs. Our analysis is divided into three parts:

1. **Segmentation** (Section 2.5.1): We investigate how various types of root models divide the keys into segments.
2. **Position Prediction** (Section 2.5.2): We analyze how accurately various combinations of models approximate the CDF.
3. **Error Bounds** (Section 2.5.3): We examine how various types of error bounds limit the error interval to be searched.

2.5.1 Segmentation

An RMI divides the keys into distinct segments based on its root model's approximation of the CDF. Assuming a root model correctly predicts the position of each key, each segment would consist of the same number of keys. Therefore, RMIs aim for an equal-depth segmentation by design. This approach to segmentation has a crucial weakness: it ignores whether the resulting segments can be accurately approximated by the next-layer models. Consequently, the quality of an RMI's segmentation cannot be assessed without considering the next layer. In contrast, other learned indexes like PGM-index [16] and RadixSpline [34], which are built bottom-up, explicitly create segments that meet a certain error tolerance.

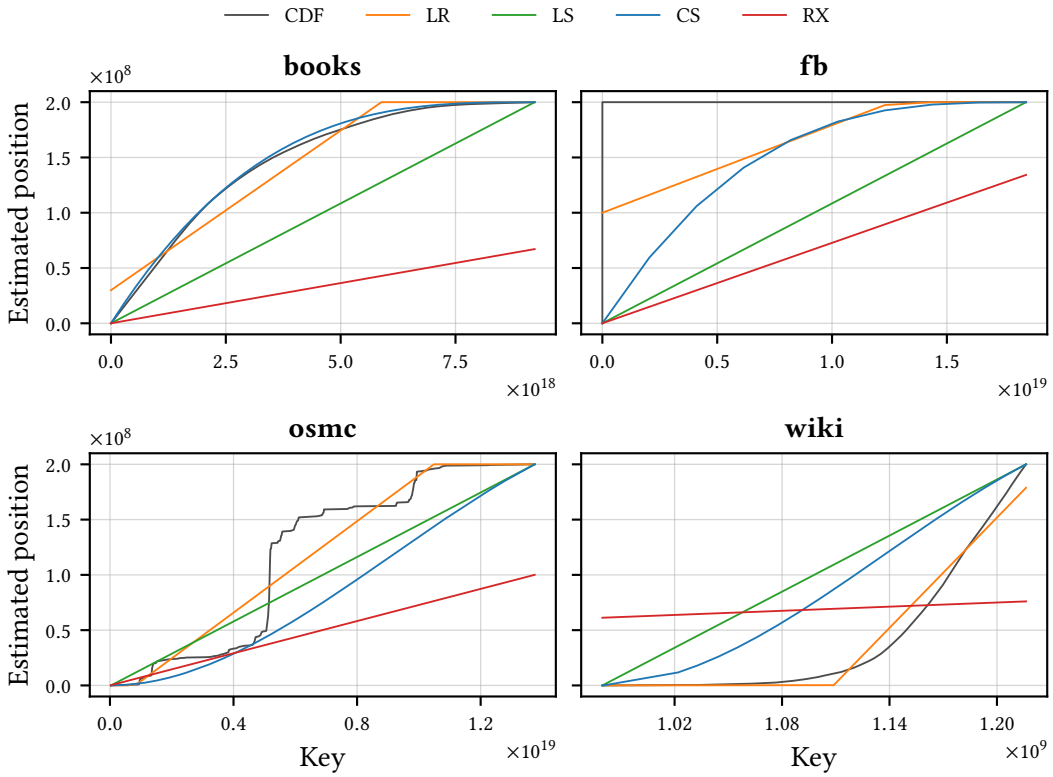


Figure 2.3: CDF and its approximation by various types of root models, used to segment the key space and distribute keys across the second-layer models.

Nevertheless, to compare the segmentation capabilities of various model types, we address two problems that may occur when segmenting keys in an RMI: empty segments, which do not contain any keys, and large segments, which contain significantly more keys than other segments. For reference, Figure 2.3 shows the CDFs and the corresponding root model approximations.

Empty Segments – Since there is a second-layer model for every segment, empty segments increase the size of an RMI without improving the prediction accuracy. Thus, we should aim for as few empty segments as possible. Figure 2.4 shows the percentage of empty segments of each model type on each dataset for a varying number of segments. We generally observe that the percentage of empty segments increases with an increasing number of segments. The more accurately a model approximates the CDF, the fewer empty segments it creates. For instance, CS produces empty segments on books only after a high number of segments is reached.

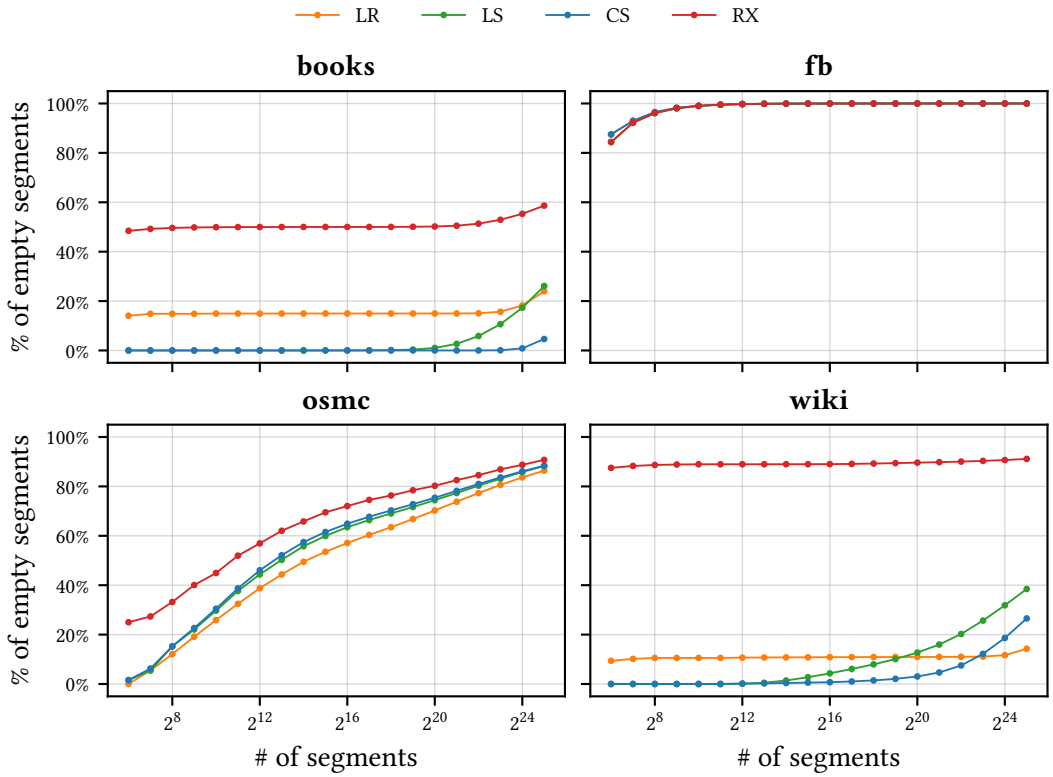


Figure 2.4: Percentage of empty segments when segmenting keys using various model types for a varying number of segments.

In contrast, radix predictions often do not cover the full range of positions, such as on the `wiki` dataset, leaving the segments associated with the non-covered positions empty. The clustered distribution of the `osmc` dataset causes percentages to be generally higher and to increase more quickly since the keys are distributed over a small number of segments. Due to the few extreme outliers that strongly affect the CDF approximation of `fb`, all models map the majority of keys to the same position, causing all of these keys to be assigned to the same segment. Increasing the number of segments gradually removes the outliers from this segment, but the single segment will continue to contain most keys.

Large Segments – Large segments contain more keys and thus potentially follow a more complex distribution, making them more difficult to approximate accurately for the second-layer models. Therefore, large partitions may negatively affect the prediction quality of an RMI. Figure 2.5 shows the number of keys that reside in the largest segment. Again, the more accurate a model approximates the CDF, the more evenly the keys are distributed over the

segments. Logically, the average segment size decreases as the number of segments increases. However, this does not necessarily apply to the largest segment.

For LR, the size of the largest partition often remains near-constant. The reason for this is that LR may produce estimates outside the range of valid positions. These out-of-range predictions are then clamped to either the first or last valid position, depending on whether they are out of range on the lower or upper end. All keys whose prediction is clamped to the same position will be assigned to the same segment. Increasing the number of segments only decreases the size of these segments until the segments consist exclusively of keys whose prediction had to be clamped.

In contrast, CS, LS, and RX do not produce estimates outside the range of valid positions and, therefore, do not exhibit this problem. As discussed before, on `fb`, almost all keys reside in a single segment, regardless of the number of segments and type of the root models. As we will see in subsequent experiments, the inability of the considered model types to segment datasets with extreme outliers is the main reason for inaccurate predictions, large error intervals, and ultimately slow lookups on `fb`.

Summary – When choosing a first-layer model type for segmentation, empty and large segments should be avoided. In our experiments, LS and CS produced the most uniform segments. RX tends to produce many empty segments. LR often creates large segments at the upper and lower end of the key space due to clamping. If none of the models satisfactorily segments the keys, as with `fb`, more complex models must be considered.

2.5.2 Position Prediction

To analyze the impact of model types on prediction accuracy, we train RMIs with all combinations of first-layer and second-layer model types and varying second-layer sizes on the four datasets. Figure 2.6 reports the median absolute error across all keys as a measure of deviation between predicted position and actual position. We decided not to report the mean absolute error because the median absolute error is more stable. Throughout the remainder, we denote an RMI that uses RX and LR in the first layer and second layer, respectively, as RX↦LR.

As expected, RMIs with more segments, and thus more second-layer models, generally produce more accurate predictions. On both the `books` and `wiki` datasets, RMIs with more than 2^{19} second-layer models even achieve errors in single digits. The `osmc` and `fb` datasets are more difficult to approximate. The `osmc` dataset has a clustered distribution that results in a high number of empty segments, leading to larger non-empty segments on average. Additionally,

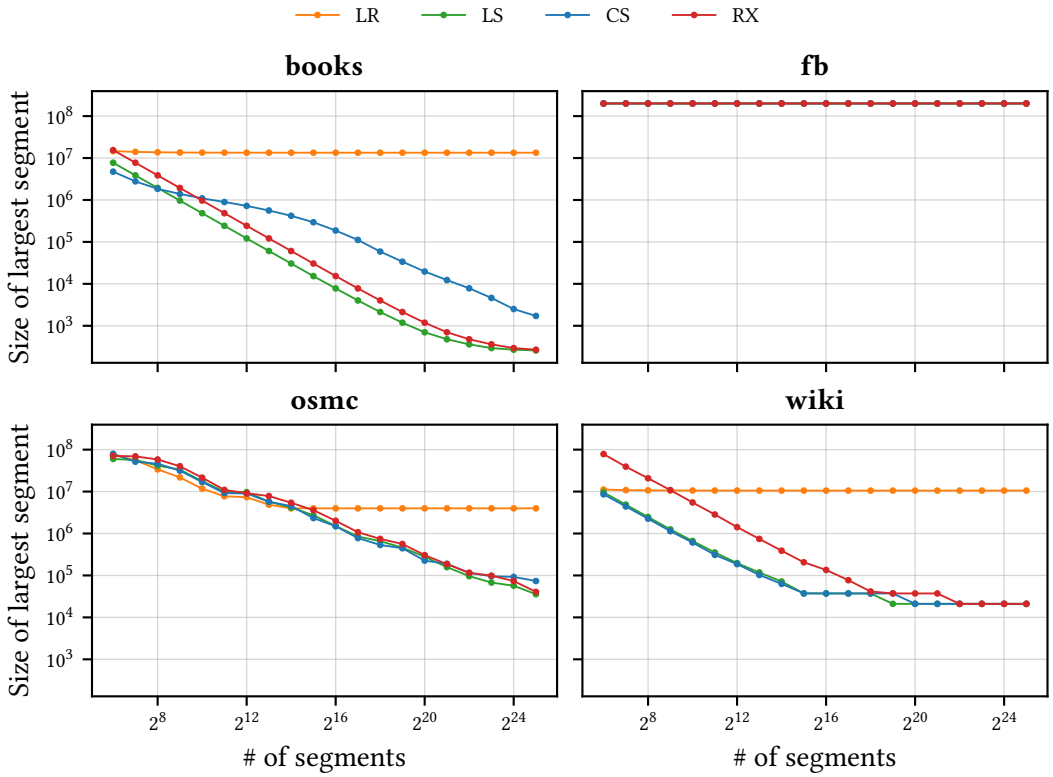


Figure 2.5: Size of the largest segment when segmenting keys using various model types for a varying number of segments.

these segments often exhibit significant noise and cannot be precisely approximated with the models considered here. Similarly, the large prediction error of `fb` can also be attributed to the single large segment. The sudden drop in prediction error between 2^{15} and 2^{17} segments occurs because fewer outliers are assigned to the large segment. Although the distribution within that large segment is close to uniform, considerable noise causes persistently high prediction errors.

Comparing the different RMI configurations, RMIs with LR, LS, and CS as root model achieve similar errors, while RX performs slightly worse. This suggests that RX is less suitable for segmentation in terms of prediction accuracy. Regarding the second-layer models, LR consistently achieves lower errors than LS. This is expected since LR is the only regression model and minimizes the mean squared error on the error bounds.

Summary – For the first layer, a segmentation that distributes the keys among many models is crucial for high prediction accuracy. In the second layer, regression models like LR achieve higher accuracy than spline models because regression models minimize the prediction error.

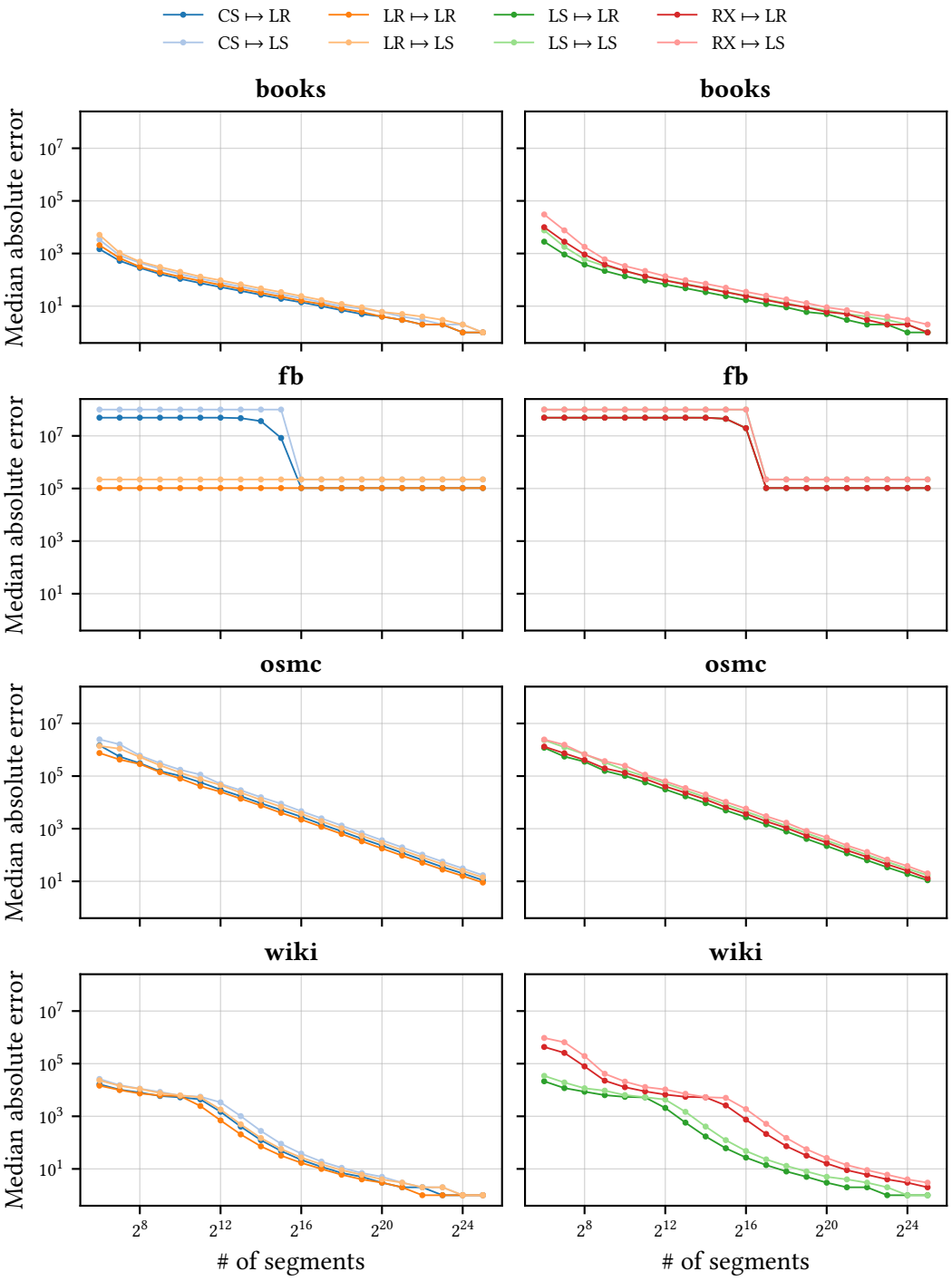


Figure 2.6: Median absolute error when predicting positions using various combinations of first-layer and second-layer model types across different segment counts.

Increasing the second-layer size of an RMI further improves its accuracy. Overall, LS↦LR and CS↦LR achieve good accuracy across datasets, except for `fb`, where poor segmentation adversely affects prediction accuracy.

2.5.3 Error Bounds

Error bounds facilitate correcting prediction errors by limiting the size of the error interval that must be searched during a lookup. To evaluate the impact of different error bounds, we again train RMIs with all combinations of first-layer and second-layer model types and varying second-layer sizes. For each configuration, we compute error bounds of different types and record the error interval sizes across all keys. In Figure 2.7, we present the median error interval size, which represents the median number of keys that have to be searched during a lookup. We focus on two combinations of models, as the observations and conclusions drawn on the error bounds apply equally to other combinations of model types.

Global bounds consistently lead to significantly larger error intervals than local bounds, even though global bounds allow for more second-layer models and achieve more accurate predictions at a similar index size. However, global bounds are susceptible to single bad predictions, whereas local bounds are more robust because they pertain to only one model. LInd and LAbs achieve similar error interval sizes. Spline models, which tend to either overestimate or underestimate, profit from LInd. LR, which often achieves similar positive and negative errors, works better with LAbs, as LAbs allows for more second-layer models at a similar index size.

Summary – When considering RMIs of similar size, local bounds consistently result in smaller error intervals than global bounds. For the preferred second-layer model type LR, LAbs achieves smaller intervals due to having more second-layer models at a similar index size.

2.6 Lookup Time Analysis

In this section, we analyze the impact of hyperparameters on the lookup performance of RMIs. Our analysis is divided into two parts:

1. **Model Types** (Section 2.6.1): We investigate the lookup performance of various combinations of first-layer and second-layer model types.
2. **Error Correction** (Section 2.6.2): We analyze the impact of various types of error bounds and search algorithms on lookup performance.

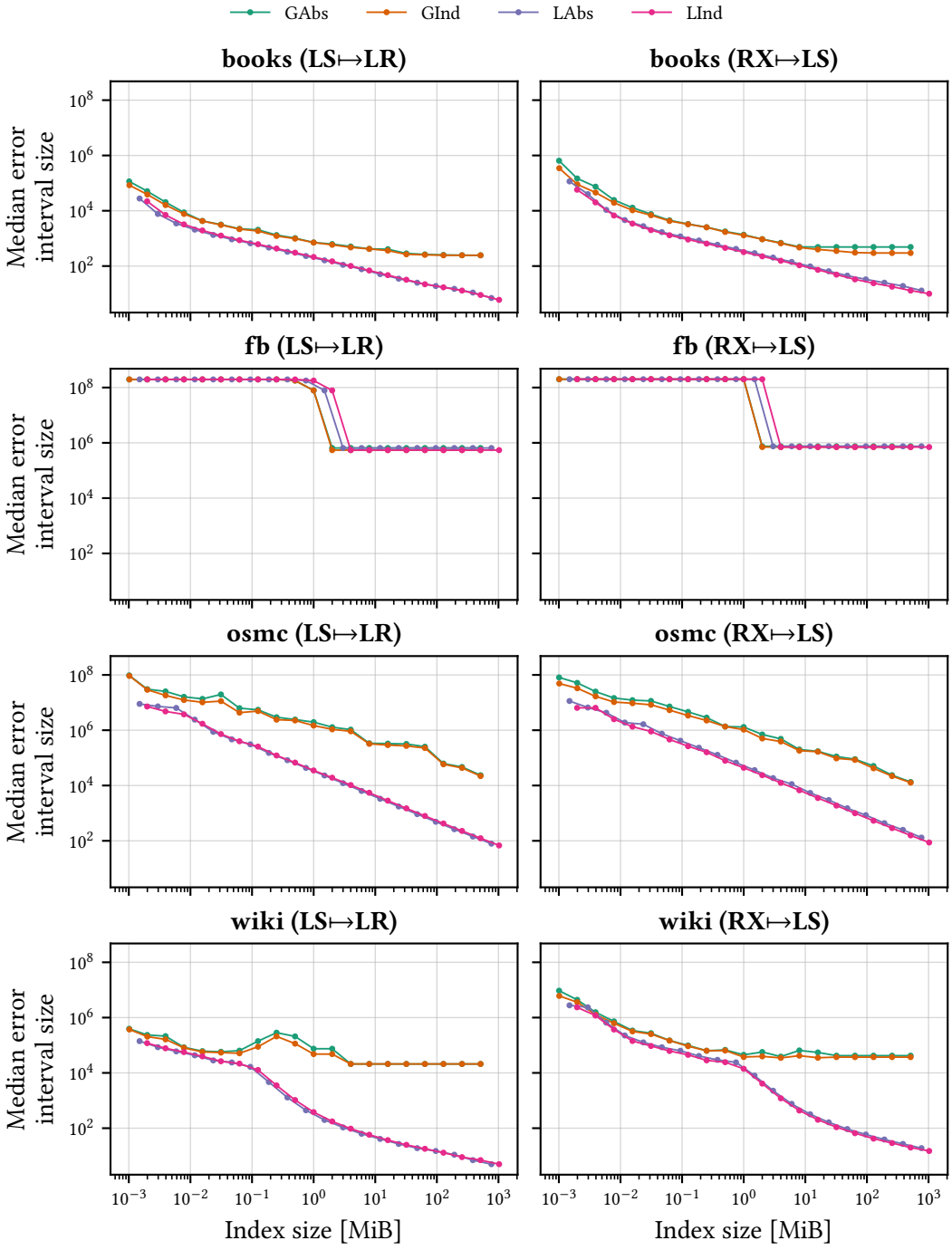


Figure 2.7: Median error interval sizes when training two combinations of first-layer and second-layer model types using various error bounds across different index sizes.

2.6.1 Model Types

To evaluate the impact of model types on lookup performance, we train RMIs with all combinations of first-layer and second-layer model types and varying second-layer sizes. We use no bounds and model-biased exponential search (NB+MExp) for error correction, as this configuration relies solely on the predictive power of the RMI and thus most clearly illustrates the differences between the various combinations of model types. In Figure 2.8, we report the average lookup time of each configuration. The dashed horizontal lines are the average time for obtaining a key using binary search.

For a fixed index size, the lookup times of different models within a dataset often differ only slightly. For instance, on `osmc` and `books`, all combinations of models have similar lookup times. However, lookup times vary significantly across different datasets. This observation is consistent with the prediction errors we saw in Section 2.5.2. The reason for this is that lookup time consists of evaluation time and error correction time. The error correction time accounts for the majority of lookup time and is determined by the prediction error. However, balancing evaluation time and error correction time is a trade-off that has to be carefully considered.

In our experiments, we only consider relatively simple models that are fast to evaluate, and as a result, there are only minor differences in evaluation time. In preliminary experiments, we also considered neural networks, which achieved higher prediction accuracy, but the faster error correction was overshadowed by a significantly higher evaluation time, ultimately resulting in considerably slower lookups. Among the models considered here, CS is the slowest to evaluate. We can observe the impact of its slower evaluation time compared to LS on `books` where, despite CS↦LR being slightly more accurate than LS↦LR, LS↦LR achieves faster lookups. Differences in evaluation time are particularly noticeable when the error correction time is relatively short, which often is the case for larger configurations.

Summary – Prediction accuracy is a strong indicator of lookup performance as it determines the error correction time. Therefore, models like CS↦LR and LS↦LR that achieve good accuracy across datasets should be chosen when optimizing lookup time. However, the more accurate the predictions are, the more important differences in evaluation time become, and models that are slightly less accurate but faster to evaluate have an advantage. Increasing the second-layer size improves accuracy and causes the lookup time to converge.

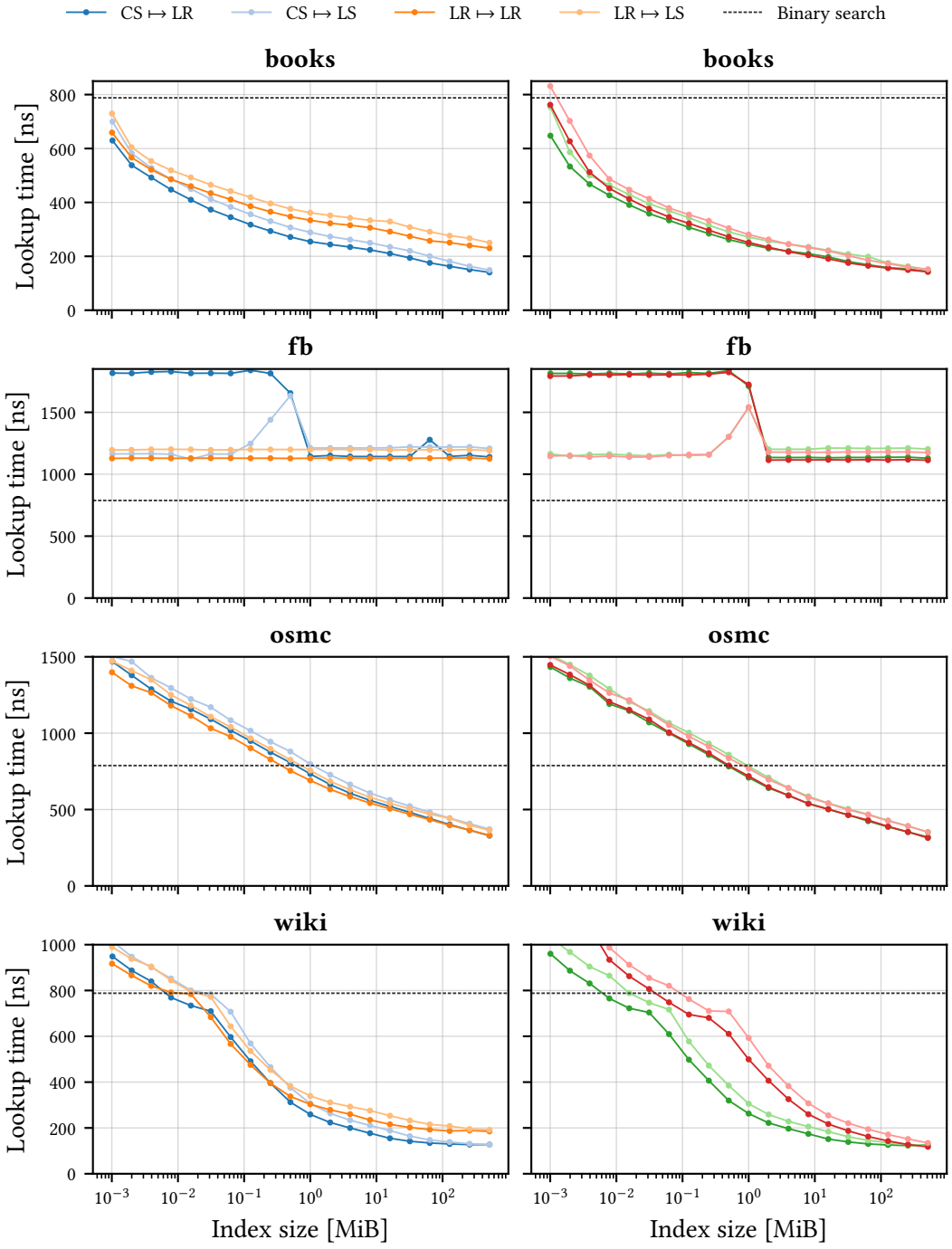


Figure 2.8: Average lookup time for various combinations of first-layer and second-layer model types with NB+MExp across different index sizes.

2.6.2 Error Correction

Next, we examine the impact of eight combinations of error bounds and search algorithms for error correction on lookup time. We consider the following combinations. NB is evaluated with MLin and MExp, as both search algorithms do not use bounds. GInd and LInd are evaluated with MBin and Bin. GAbs and LAbs are evaluated with Bin only, as MBin and Bin are the same in case of absolute bounds, as argued in Section 2.4.2.

In Figure 2.9, we report the average lookup time, showing two representative combinations of first-layer and second-layer model types. However, our observations on various combinations of error bounds and search algorithms also apply to the other combinations of model types.

We observe that configurations with either local bounds or no bounds perform best. Local bounds generally perform better than global bounds, which is consistent with our observation from Section 2.5.3. Nevertheless, binary search mitigates differences in error interval size drastically. For instance, global and local bounds perform almost identical with LS↦LR on `books`, although the error interval sizes differ by more than an order of magnitude. LInd and LAbs also perform almost identical, with a maximum performance difference of factor 1.1x. As observed in Section 2.5.3, LS performs better with LInd as it tends to either overestimate or underestimate, whereas LR performs better with LAbs as its loss function balances overestimations and underestimations. Considering LInd, there is hardly any difference between Bin and MBin.

As observed in Section 2.6.1 in regard to model types, the choice of error bounds not only affects error correction time but also evaluation time, as error bounds induce overhead for computing the error interval's limits. Hence, RMIs without error bounds are faster to evaluate. The faster evaluation is particularly noticeable when RMIs achieve high prediction accuracy and thus fast error correction. In these cases, NB+MExp performs better than configurations with bounds, as can be seen with `books` and `wiki`.

To further analyze when to use NB+MExp over configurations with bounds, we also recorded the mean $\log_2$ error as an estimate of the number of search steps required by MExp. Starting with a mean $\log_2$ error of around 7 to 10, NB+MExp is faster than LAbs+Bin. NB+MLin requires even lower errors to be similarly fast.

Summary – The best combination of error bounds and search algorithm depends on the predictive accuracy of the RMI. If the mean $\log_2$ error is sufficiently small, NB+MExp achieves the best lookup times due to being faster to evaluate. For larger errors, configuration with local bounds, such as LAbs+Bin, perform better.

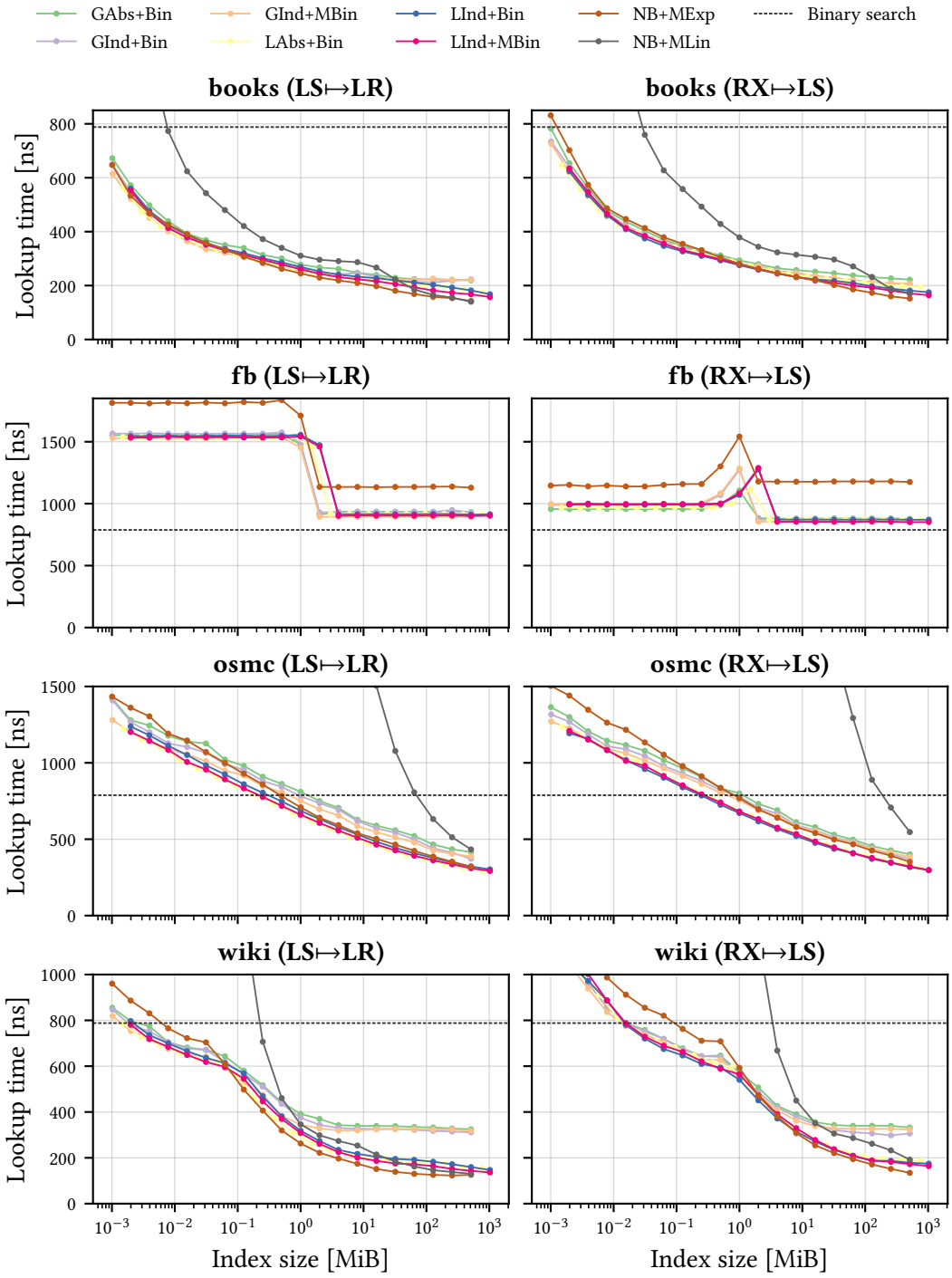
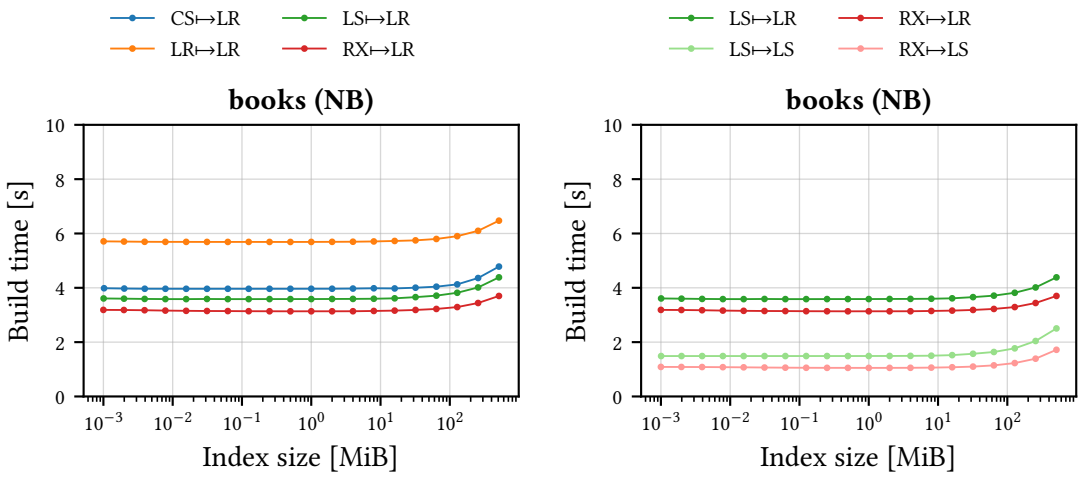


Figure 2.9: Average lookup time for two combinations of first-layer and second-layer model types with various combinations of error bounds and search algorithms across different index sizes.



(a) First-layer type

(b) Second-layer type

Figure 2.10: Build times when training various first-layer and second-layer model types with NB on books across different index sizes.

2.7 Build Time Analysis

In this section, we analyze the build time of our implementation of RMIs and compare it with the reference implementation [47]. Recall that the build process of a two-layer RMI consists of four steps: (1) training the root model, (2) creating segments, (3) training the second-layer models, and (4) computing error bounds. Figure 2.10 and 2.11 only show build times on books, as build times are largely independent of the dataset, except for minor caching effects on large configurations. We discuss each aspect that affects build time individually below.

First-Layer Type – Consider Figure 2.10a for a build time comparison of different root models. Models in general, and root models in particular, not only differ in training time, which affects step (1), but also in evaluation time, which affects steps (2) and (4). The most notable difference between the models in terms of training time is whether a model considers all keys, like LR, or a constant number of keys, like LS, CS, and RX. Since the evaluation time of LR and LS is the same, the difference in build time in Figure 2.10a can be attributed entirely to the training time of the root model. Like LS, RX also considers only two keys for training. Here, the faster build time of RX is caused by the faster evaluation of RX during segmentation. CS is faster than LR because it again only considers a constant number of keys but slower than LS because training and evaluation are slightly slower.

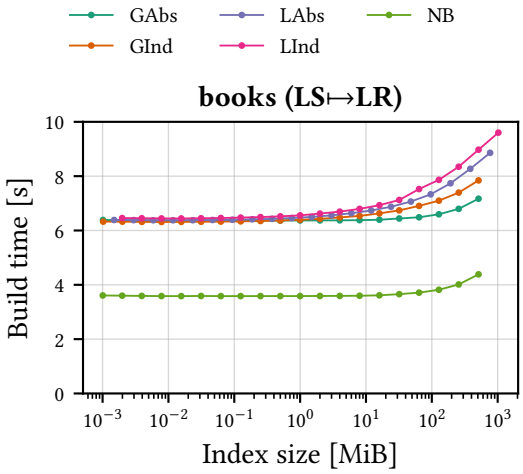
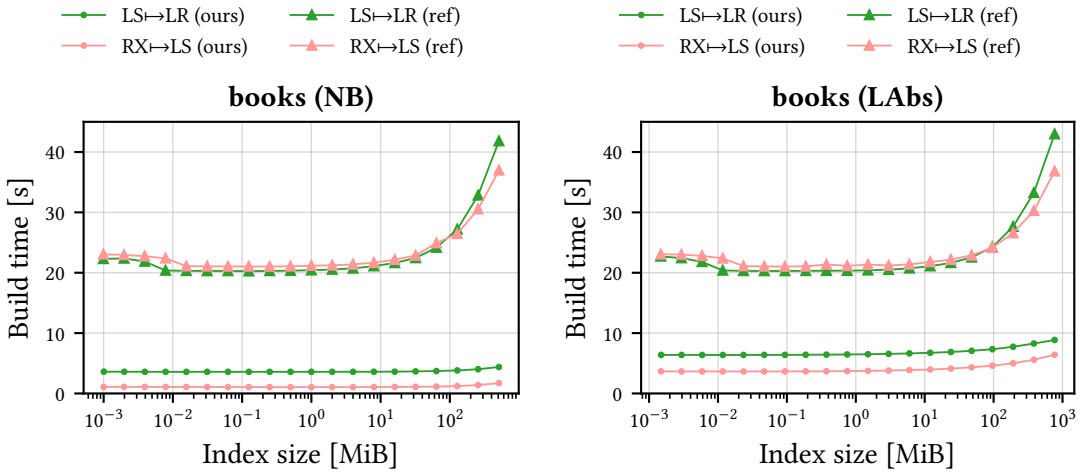


Figure 2.11: Build times when training LS↦LR on `books` using various error bounds across different index sizes.

Second-Layer Type – Consider Figure 2.10b for a build time comparison of different second-layer models. Analogous to the root model type, the second-layer model type affects training time and evaluation time. Second layers consisting of LS models take about two seconds less to train than second layers consisting of LR models. As we do not compute bounds here, the second layer is never evaluated. Otherwise, evaluation time would be the same for LR and LS.

Error Bounds – Consider Figure 2.11 for a build time comparison of different error bounds. Computing error bounds requires evaluating the RMI on every key plus the actual computation of the bounds. This additional effort explains the difference in built time between NB and configurations with bounds. The difference between individual configurations with bounds is mainly due to branch misses when calculating the bounds. At a similar index size, local bounds trigger more branch misses than global bounds, and individual bounds trigger more branch misses than absolute bounds.

Index Size – Consider again the RMI configuration without bounds in Figure 2.11. The build time remains almost constant as long as the entire RMI fits in cache (20 MiB). Once the RMI no longer fits in cache, the build time increases due to cache misses. Next, consider the configurations with bounds in Figure 2.11. Here, the previously described branch and cache misses add up, and the build time already increases for configurations that are smaller than the cache size. The increase in build time is less pronounced if a configuration produces many empty segments due to less cache misses.



(a) No bounds

(b) Local absolute bounds

Figure 2.12: Build times when training two combinations of first-layer and second-layer model types with NB and LAbs on books across different index sizes, both using our implementation (ours) and the reference implementation (ref).

Reference Implementation – Figure 2.12 shows build times of our implementation (ours) and the reference implementation (ref). Figure 2.12a and Figure 2.12b compare configurations with NB and LAbs, respectively. Build times for both types of bounds are almost identical for the reference implementation because the reference implementation always computes bounds during training and only decides later whether these computed bounds are kept or discarded. Considering only configurations with LAbs, our implementation improves build times by 2.5x to 6.3x. We attribute this improvement to our optimized segmentation for monotonous root models, which avoids copying keys as described in Section 2.4.1.

Summary – RMIs can be built in a matter of seconds. For a given combination of models, the build time remains almost constant as long as the RMI fits in the cache. The computation of error bounds leads to additional cache and branch misses, which negatively impact build times.

2.8 Configuration Guideline

Based on our findings from the previous sections, we present a compact guideline for configuring RMIs. Our guideline does not guarantee to always suggest the optimal configuration in terms of lookup time, but it is easy to follow and achieves competitive lookup performance.

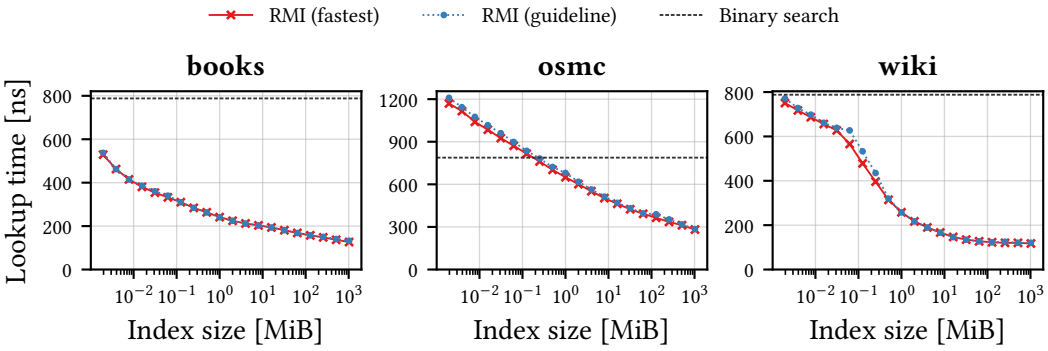


Figure 2.13: Average lookup time when performing lookups with RMIs configured using exhaustive enumeration (fastest) and our guideline (guideline) across different index sizes.

Additionally, we address limitations of our guideline. Given a maximum allowed index size budget, we propose to configure RMIs as follows.

Model Types – LS↦LR with the largest feasible second-layer size within the allocated budget. CS and LS both segment most datasets well, but we choose LS as it is slightly faster to train and evaluate. Although more accurate predictions can be obtained with CS, CS is only faster for small RMIs, where the improvement in search time outweighs the longer evaluation time. LR as second-layer model minimizes the error and thus always performs better than LS. Larger RMIs generally achieve smaller errors and thus perform better, which is why we choose the maximum number of second-layer models within the budget.

Error Correction – Choose between LAbs+Bin or NB+MExp based on the prediction accuracy. Our experiments show that LAbs+Bin performs better than NB+MExp until a certain error threshold is reached. This error threshold is hardware-dependent and must be determined empirically once. We use the mean $\log_2$ error as measure of error to estimate the number of search steps with exponential search and determine the error threshold to be 5.8 on our hardware. Whenever the mean $\log_2$ error of our RMI with NB is below that threshold, we use NB+MExp; otherwise, we use LAbs+Bin.

Figure 2.13 compares the lookup times of configurations obtained by our guideline with the fastest configurations obtained using exhaustive enumeration. As before, we omit `fb` as none of the considered models segments `fb` well. We consider size budgets between 2 KiB and 1 GiB. Our guideline is on average only 2.0% slower than the fastest configuration with a maximum performance decline of 11.3% on `wiki`.

Implementing our guideline requires training a maximum of two RMIs and entails the following three steps:

1. Train an RMI with LS↦LR and NB within the allocated budget.
2. Calculate the mean $\log_2$ error of this RMI across all keys.
3. If the error exceeds the threshold, train and use an RMI with LAbs within budget. Otherwise, use the existing RMI.

Limitations – In order to be simple and induce as little overhead as possible, our guideline neglects some aspects that are required for optimal configuration.

1. **Fixed Model Types**: Our guideline uses fixed model types. While LS↦LR strikes a good balance between fast training and accurate predictions for datasets without outliers, datasets containing outliers require a more appropriate first-layer model.
2. **Fixed Error Correction**: Our guideline selects between LAbs+Bin and NB+MExp based on a rough estimate of expected search steps. While LAbs+Bin generally performs well and NB+MExp excels when prediction accuracy is high, there are scenarios, where alternative error correction strategies prove to be marginally faster.

2.9 Comparison With Other Indexes

In this section, we compare our implementation of RMIs with the indexes introduced in Section 2.4.5, varying the parameters listed in Table 2.4 to obtain indexes of various sizes. Configurations of our RMI implementation are determined based on our guideline, while the configuration of the reference implementation are chosen based on its optimizer [47]. The comparison is divided into two parts:

1. **Lookup Time** (Section 2.9.1): We assess the lookup performance of various indexes across both the four original datasets and scaled-down versions. Additionally, we analyze the breakdown of lookup time into model evaluation and search time.
2. **Build Time** (Section 2.9.2): We examine the build time of various indexes across the four datasets, varying the size of the indexes to assess how build time scales with index size.

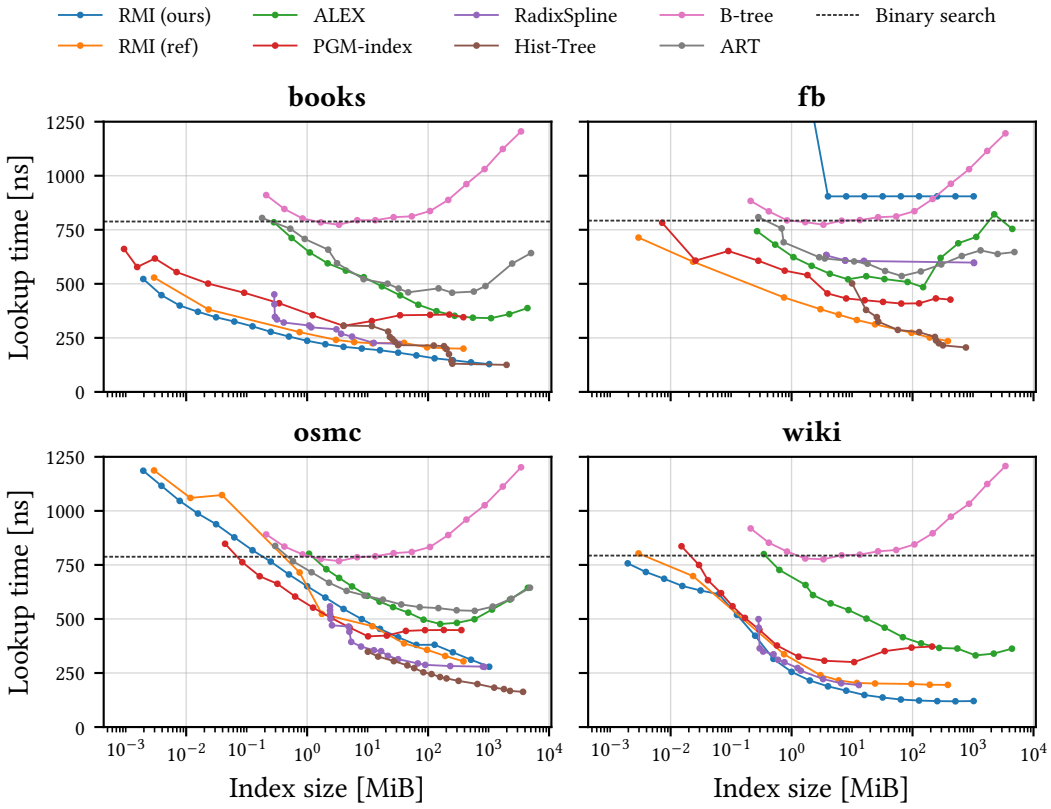


Figure 2.14: Average lookup time for various indexes across different index sizes.

2.9.1 Lookup Time

Original Datasets – We first compare lookup times on the original datasets with respect to index size. During a lookup, each index yields a search range, either through error bounds or level of sparsity. We use binary search to find keys in that search range. In Figure 2.14, we report average lookup times. For indexes with multiple hyperparameters, such as RadixSpline and Hist-Tree, we present Pareto-optimal configurations in terms of index size and lookup time to enhance readability. As a result, the number of data points shown differs across datasets for these indexes. Furthermore, Hist-Tree and ART do not support duplicates and are therefore not evaluated on `wiki`. Overall, our results are consistent with previous reports [33, 49, 29].

Let us first consider the traditional indexes. Hist-Tree is the fastest index on all datasets except `wiki`, where it cannot be applied due to duplicates. However, Hist-Tree requires index sizes of 100 MiB and more to reach its full potential. The best-performing configurations of Hist-Tree use high branching factors, resulting in few levels, while achieving error intervals of less than

64 keys. B-tree is the only index whose performance is completely independent of the data distribution but also the slowest index, barely beating binary search. ART is always faster than B-tree but noticeably slower than all learned indexes except for ALEX.

The performance of learned indexes highly depends on the data distribution. Learned indexes achieve the fastest lookup times up to a certain index size, beyond which Hist-Tree outperforms the other indexes. This implies that learned indexes perform particularly well compared to traditional indexes when limited space is available and space-efficient configurations are required.

On `books`, `fb`, and `wiki`, either our implementation or the reference implementation of RMIs dominates the other learned indexes across index sizes. On `osmc`, RadixSpline performs better than RMIs, whereas PGM-index outperforms RMIs only at smaller index sizes. ALEX is clearly the slowest learned index, which can be attributed to its more complex and adaptive structure.

Next, we compare our implementation and the reference implementation of RMIs [47]. On `books` and `wiki`, our implementation dominates the reference implementation despite using our guideline. There are two reasons for the improved performance of our implementation:

1. **Neglecting Evaluation Time**: Unlike the optimizer described [47], the publicly available implementation [45] does not consider evaluation time in its optimization process. Instead, the optimizer selects configurations solely based on the lowest mean $\log_2$ error. While this results in selecting the configuration with the fastest error correction time, it does not guarantee to select the configuration with the fastest overall lookup time. The configurations chosen by our guideline consistently have fast evaluation times at the cost of potentially slower error correction.

2. **Fixed Error Bounds**: The optimizer of the reference implementation always picks LAbs. Our experiments in Section 2.6.2 show that for accurate RMIs, NB+MExp performs better, which is considered by our guideline.

On `osmc`, no implementation dominates the other. Here, RMIs are never sufficiently accurate for our guidelines to deviate from LAbs+Bin. Thus, differences in performance are solely due to the choice of model types.

On `fb`, the reference implementation clearly dominates our implementation. As discussed before, LS is not suitable for segmenting datasets with extreme outliers. Here, the reference implementation selects a variant of LR that ignores the lowest and highest 0.01% of keys during training. This approach effectively eliminates the outliers in `fb` from the segmentation process

but assumes that there are at most 0.01% of outliers at either end of the key space. We did not include this model type in our evaluation because we believe that a more robust solution to segmentation should be sought.

Scaled-Down Datasets – Next, we analyze the lookup performance on smaller datasets. For this purpose, we reduced the size of the original datasets in steps by powers of ten through downsampling and determined the best-performing configuration for each index on each of the scaled-down datasets. The average lookup times are presented in Figure 2.15. It is important to note that a scale factor of 10^0 represents the original, unscaled dataset. The symbol $\times$ indicates instances where the corresponding index was not supported, either due to the presence of duplicates (Hist-Tree and ART on the `wiki` dataset) or an optimizer crash (RMI reference implementation on the `fb` dataset).

We anticipated identifying a threshold in dataset size beyond which using RMIs over binary search would cease to be beneficial. Surprisingly, our implementation of RMIs consistently outperforms binary search, even on very small data sets that fit entirely into the L1 cache, despite the RMI exceeding the cache capacity. We attribute this to two factors.

1. **High Prediction Accuracy**: On smaller datasets, the prediction accuracy of RMIs is exceptionally high, resulting in minimal or no need for error correction through search.
2. **Optimal Caching**: The repeated use of the same RMI allows the CPU to optimize caching of the relevant parts of the RMI.

We do not expect RMIs to outperform binary search on small datasets with cold caches. However, consistently measuring cold cache performance is challenging and outside the scope of this work.

Comparing our RMI implementation, configured using our guideline, and the reference implementation, configured using its optimizer, our implementation consistently outperforms the reference implementation, except for the non-scaled version of the `fb` dataset. This finding reinforces our assertion that optimizing search time alone is insufficient for optimal RMI configuration. Additionally, it indicates that outliers are the primary cause of poor performance on the `fb` dataset. The performance aligns with that of other datasets once the scaled-down versions of the `fb` dataset no longer include outliers, which occurs at scale factors of 10^{-2} and smaller.

All learned indexes outperform binary search on all but the smallest datasets, where PGM-index, ALEX, and RadixSpline are slightly slower. Overall, Hist-Tree is the fastest index, either

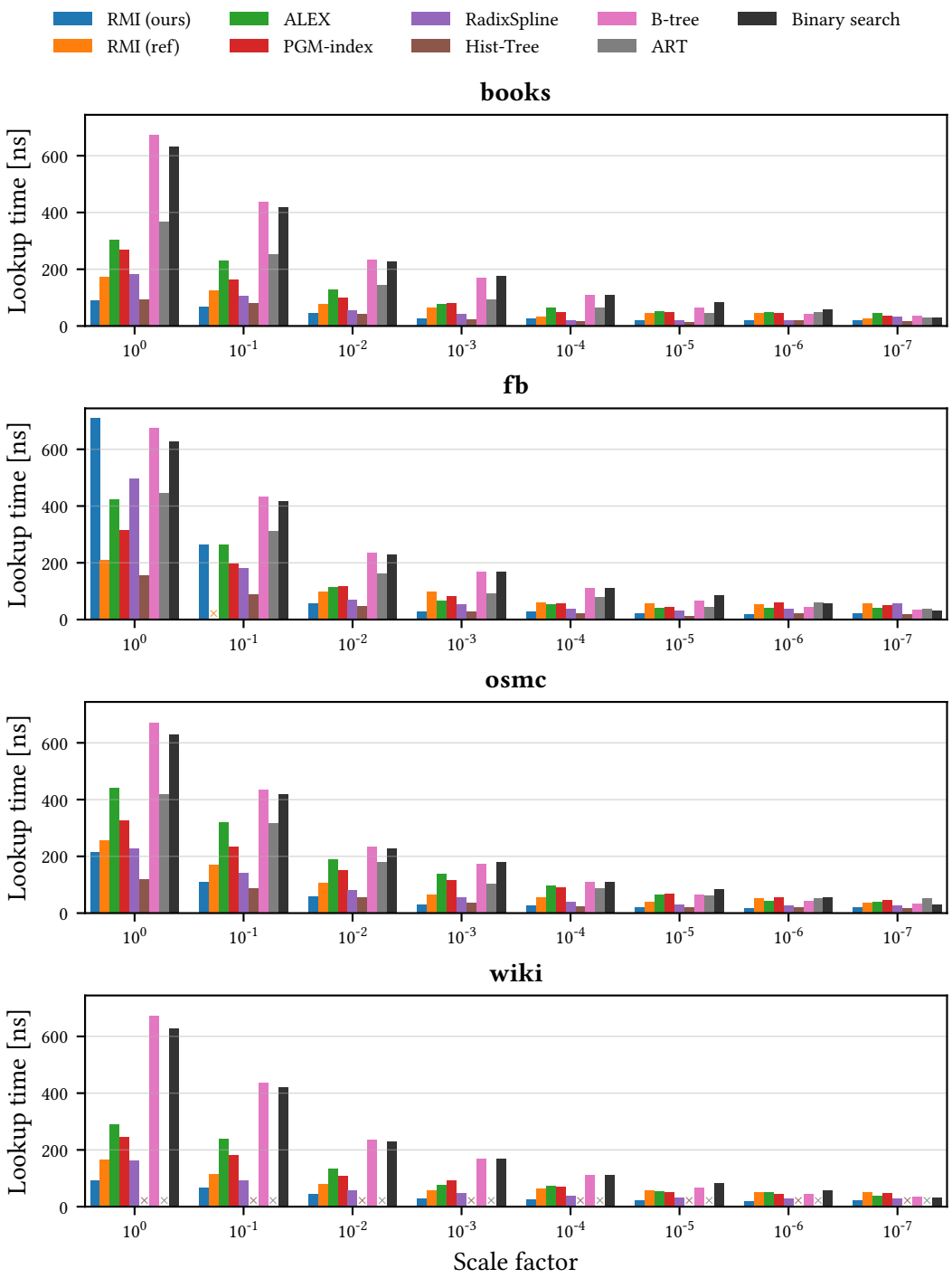


Figure 2.15: Average lookup time for the best-performing configuration of various indexes on scaled-down datasets.

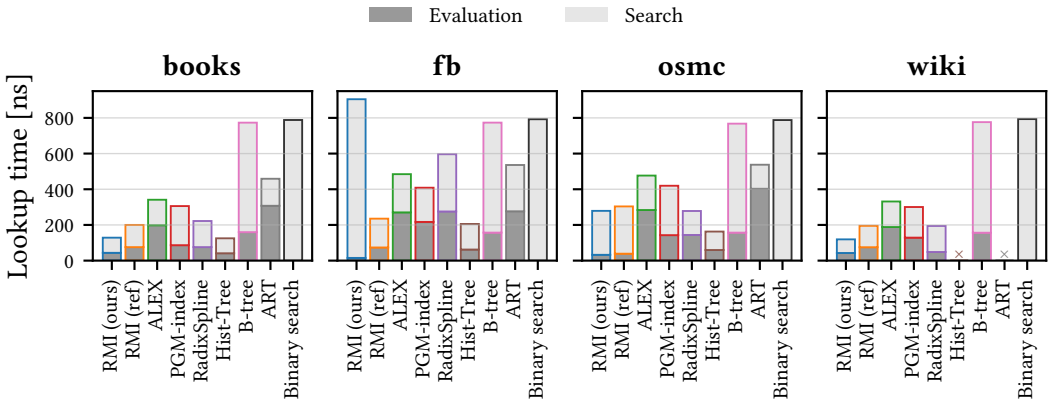


Figure 2.16: Lookup time breakdown for the best-performing configuration of various indexes.

matching or outperforming RMIs on datasets without duplicates. Hist-Tree is notably more space-efficient than RMIs, particularly on datasets with 20,000 or fewer keys, where Hist-Tree occupies less than 1 MiB of space compared to tens of MiB required by the RMIs. RadixSpline is consistently slower than RMIs but, apart from the non-scaled version of the `fb` dataset, faster than PGM-index and ALEX. ALEX and PGM-index are clearly inferior to the other learned indexes.

Considering traditional indexes, B-tree outperforms binary search only on datasets with a scaling factor between 10^{-6} and 10^{-5}, which equals 200 to 2000 keys. On datasets with more than 2000 keys, ART is faster than B-tree. On smaller datasets, B-tree is faster than ART. Both ART and B-tree are generally slower than the top-performing learned indexes on the scaled-down datasets.

Lookup Time Breakdown – Let us now break down lookup time into evaluation time (evaluating the model or traversing the tree) and search time (searching within the error interval or data page). Figure 2.16 shows the average lookup time for the best-performing configuration of each index divided into evaluation and search time on the original datasets.

There is a trade-off between fast evaluation and fast search. RMIs clearly prioritize fast evaluation: The prediction leads to the correct segment in a fixed number of steps. However, RMIs do not provide any guarantees on the prediction accuracy, resulting in potentially slow error correction times. Adding more segments by increasing the number of second-layer models continuously improves the lookup performance because more segments hardly increase the evaluation time while simultaneously improving the search time. If the evaluation time exhibited by our implementation is significantly faster than that of the reference implementa-

tion, it is not only due to differences in the selected models but primarily because the guideline selected a configuration without bounds.

In contrast, PGM-index and RadixSpline prioritize fast error correction: Both indexes limit the maximum error at the cost of a slower evaluation that requires traversing multiple layers or performing intermediate searches. At a certain threshold, the reduced search time of a smaller maximum error does not outweigh the longer evaluation time, resulting in worse overall lookup performance. Thus, despite having fewer hyperparameters than RMIs, achieving optimal configurations for PGM-index and RadixSpline is an elaborate task.

Summary – Our analysis demonstrates that learned indexes generally outperform binary search. However, Hist-Tree emerges as the fastest index. Furthermore, our RMI implementation consistently outperforms the reference implementation, underscoring the importance of comprehensive optimization beyond search time alone. Traditional indexes like B-tree and ART typically trail behind learned indexes in terms of both performance and space-efficiency.

2.9.2 Build Time

In the following, we compare the indexes in terms of build time. In Figure 2.17, we report build times for the index configurations evaluated in terms of lookup time in Section 2.9.1. We show the raw build times without the time required to determine hyperparameters, such as by running the reference implementation's optimizer [47] or determining Pareto-optimal configurations of RadixSpline and Hist-Tree. Some indexes require data preparation to be built. For instance, ALEX, B-tree, and ART are not only built on the keys but also explicitly require the positions to which these keys should be mapped. Since these preparation steps could be circumvented by a specialized implementation that implicitly computes this mapping during bulk loading, we do not consider them part of the build time.

The index size of B-tree, ART, and ALEX is determined by the level of sparsity. In contrast to learned indexes, these indexes are built on a subset of the keys and therefore provide fast build times, particularly at smaller index sizes. However, at larger index sizes, the build time increases considerably, as with an increasing number of keys, the structure of these indexes becomes more complex, introducing more levels.

In contrast, RMI, PGM-index, and RadixSpline are constructed using the entire dataset, leading to inherently higher build times. RMI and RadixSpline have a fixed number of layers. Therefore, their build time is hardly impacted by the data distribution. Their build time only increases once the index does not fit into cache anymore. The sudden decrease in build time of RMIs on books

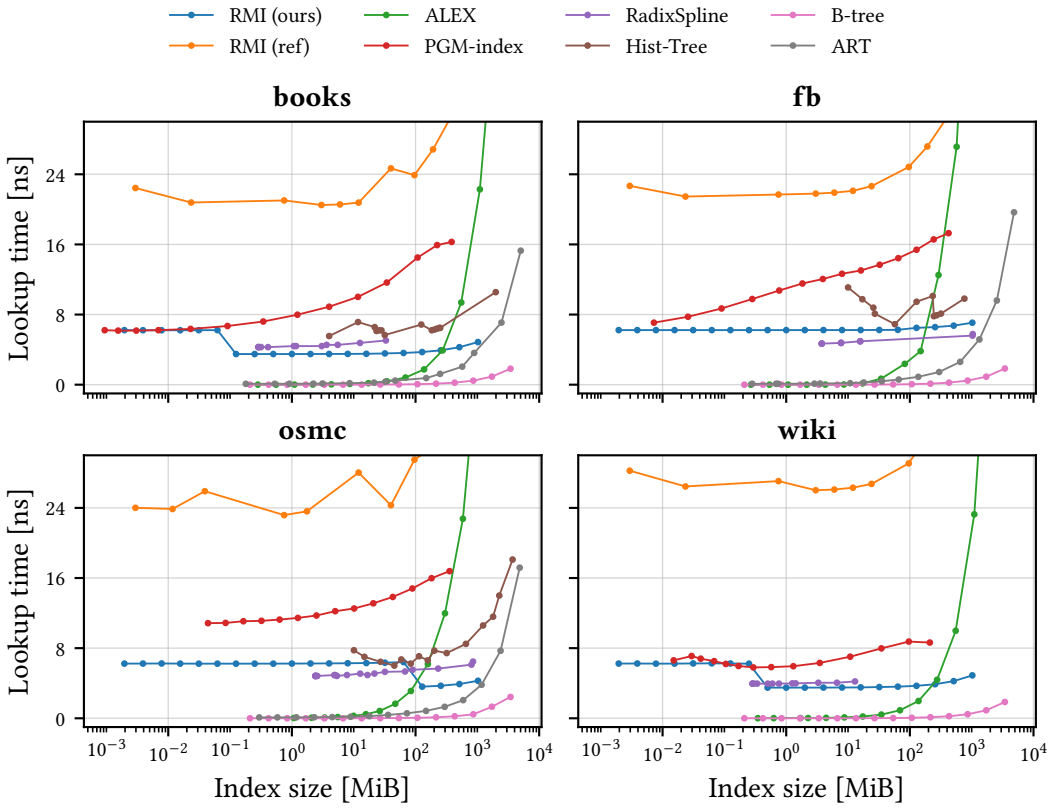


Figure 2.17: Build times for various indexes across different index sizes.

and `wiki` is caused by the guideline selecting a configuration without bounds, which is faster to build. PGM-index, on the other hand, has a variable structure, in terms of both number of layers and segments. Depending on the distribution and the desired error, more layers have to be trained, leading to a steeper increase in build times compared to RadixSpline and RMI.

Differences in build time between our RMI implementation and the reference implementation were discussed in Section 2.7. The reference implementation's build time varies due to the optimizer selecting different models. Hist-Tree initially shows similar build times to learned indexes, but its build time increases rapidly as its size grows due to a deeper structure.

Summary – Learned indexes offer superior lookup performance but require significantly longer build times. Hence, enhancing build efficiency should be a future priority.

2.10 Conclusion and Future Work

We provided an extensible open-source implementation of RMIs and conducted a comprehensive hyperparameter analysis of RMIs focusing on prediction accuracy, lookup time, and build time. Based on this analysis, we developed a simple-to-follow guideline for configuring RMIs, which achieves competitive performance. Additionally, we improved the build time of RMIs by leveraging the monotonicity of models, thus avoiding the need to copy keys during assignment to second-layer models.

In the future, we plan to extend our implementation to also support multi-layer RMIs and additional model types. Addressing the segmentation of datasets with extreme outliers is also a priority. These enhancements aim to further optimize the performance and applicability of RMIs across various datasets and use cases.

Chapter 3

Index Access Strategies for Index Scans

3.1 Introduction

In modern database systems, efficient query processing is essential for extracting insight from ever-increasing amounts of data. As databases grow in size and complexity, minimizing query execution time becomes critical for applications ranging from real-time analytics to large-scale data warehousing. To achieve this, database systems rely on different query engine architectures, which fall into two fundamental approaches: interpretation and compilation.

Interpreting query engines process queries directly by mapping each operation to a set of predefined instructions. This approach provides flexibility, as execution can adapt to runtime observations, allowing for optimizations that are impossible ahead of time. However, interpretation incurs overhead from frequent function calls and dynamic type checks. Compiling query engines, in contrast, generate specialized code for a given query, which is then optimized and executed. This approach improves performance by eliminating interpretation overhead. However, once compiled, the query plan is fixed, leaving no opportunity to adjust execution based on runtime information.

This lack of flexibility in compiled query execution limits optimization opportunities. Certain information, such as the actual number of qualifying tuples in a selection, is only available at runtime, yet traditional compiling engines cannot incorporate such insights into the generated code. To address this limitation, we introduce a novel query engine architecture that allows

partial execution of query plans during compilation. By interpreting selected operations at compile time, the engine can collect runtime observations and embed them into the generated code, opening opportunities for additional optimizations and potentially improving overall query performance.

A promising candidate for exploring this architecture is the index scan, a fundamental database operation that consists of multiple steps. Determining which of these steps can benefit from partial execution offers an opportunity to evaluate the effectiveness of this approach in practice.

Contributions – We investigate how integrating interpretation into a compiling query engine can improve index scan performance. Specifically, we make the following contributions:

1. We present a novel compiling query engine architecture that allows the query compiler to partially execute query plans, collect runtime observations, and incorporate then into the generated code for improved execution.

2. We introduce three distinct index access strategies based on our novel query engine architecture, each determining which steps of an index scan are executed during query compilation versus at runtime. Additionally, we demonstrate their applicability across two execution environments.

3. We implement these strategies in the modern database system mutable, an in-memory database system currently under development in our group.

4. We conduct a comprehensive experimental analysis to compare the three strategies in terms of performance, aiming to discern the optimal conditions for each strategy's application. Further, we investigate multiple variants of each strategy and assess whether the strategies can benefit from caching compiled plans of previous queries.

Outline – The remainder of this work is structured as follows. Section 3.2 provides background on query processing in database systems, focusing on execution by interpretation and compilation. In Section 3.3, we present a novel compiling query engine that enables the query compiler to partially execute QEPs during compilation. Based on this query engine, we introduce three strategies for accessing indexes in the context of an index scan in Section 3.4 and explore technical aspects of our implementation in Section 3.5. Our experimental evaluation is presented in Section 3.6. We summarize related research in Section 3.7 and conclude in Section 3.8.

3.2 Query Processing

This section introduces fundamental aspects of query processing within a database system. It covers the processing pipeline from submitting a query to obtaining the query result (Section 3.2.1) and explores table access methods with a focus on table scan and index scan (Section 3.2.2). In concludes with a discussion on query execution methods, encompassing both interpretation and compilation (Section 3.2.3), and the differences between integrated and isolated execution environments (Section 3.2.4).

3.2.1 Processing Pipeline

Processing a query written in Structured Query Language (SQL) involves several sequential steps, illustrated in Figure 3.1. In the initial step, the query is checked for syntactical correctness. This is achieved by running the query through a lexer and parser, resulting in an abstract syntax tree (AST). Subsequently, semantic analysis is performed to verify that the query is semantically correct. During this analysis, identifiers such as table and column names are associated with the corresponding object in the database schema, and the AST is annotated with additional details such as type information.

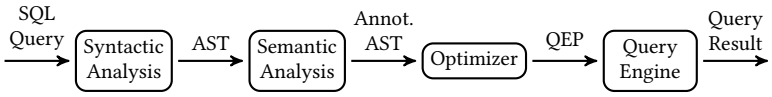
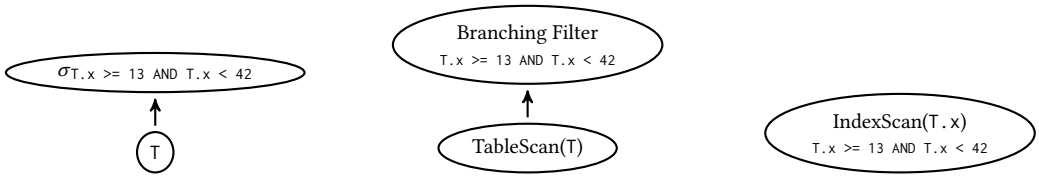


Figure 3.1: Overview of a generic query processing pipeline in a database system.

Following semantic analysis, query optimization is undertaken to determine the most efficient execution plan. The optimization process generally involves two main stages. In the first stage, the optimizer computes a join order and applies a set of optimization rules to obtain a logical query plan [23]. A logical query plan represents the high-level, implementation-independent sequence of operations required to execute a query. Typically, logical query plans are depicted as directed acyclic graphs, where nodes represent relational algebra operators such as selection or join [8], and edges illustrate the flow of data. In the second stage, the logical query plan is transformed into an efficient physical plan, also known as the query execution plan (QEP). The QEP specifies the sequence of operations chosen by the optimizer for efficiently executing the query in the database system. This transformation involves selecting the appropriate physical implementation of logical operators, like joins, and access paths to minimize resource usage and execution time, thus ensuring optimal performance of the query.



(a) Logical query plan (b) QEP with table scan (c) QEP with index scan

Figure 3.2: Logical query plan and two potential QEPs for the query in Listing 3.1.

Finally, the QEP is handed over to the query engine, which executes the plan and returns the query result. Query engines generally fall into two categories: those that execute QEPs by interpreting them and those that execute QEPs by compiling them. We explain both approaches in more detail in Section 3.2.3.

Listing 3.1: An example query to illustrate query processing in a database system.

```
1 SELECT *
2 FROM T
3 WHERE T.x >= 13 AND T.x < 42;
```

Example – Consider the SQL query shown in Listing 3.1, which retrieves all tuples from table `T` where attribute `T.x` is greater than or equal to 13 and less than 42. Figure 3.2a illustrates the logical query plan for this query, where a selection operator (σ) applies the range predicate (`T.x >= 13 AND T.x < 42`) to the tuples in table `T`.

Assuming there exists an index on attribute `T.x`, the optimizer has two options for transforming the logical query plan into a QEP: a table scan with a filter operator (Figure 3.2b), or an index scan (Figure 3.2c). In the QEP illustrated in Figure 3.2b, the optimizer selects a branching implementation for the filter operator. Although this is a common choice, alternatives, like predicated execution, are also viable. The subsequent section provides a detailed explanation of the two table access methods, table scan and index scan.

3.2.2 Table Access Methods

Transforming a logical query plan into a QEP involves selecting suitable access methods for retrieving tuples from the tables. Factors such as query predicates, index availability, and cost estimates play a crucial role in this decision. In the following, we take a closer look at two commonly used access methods: table scan and index scan.

Table Scan – In a table scan, all tuples of a table are sequentially loaded from the store. Filter predicates are applied to each tuple only after it has been loaded. The table scan is a universal

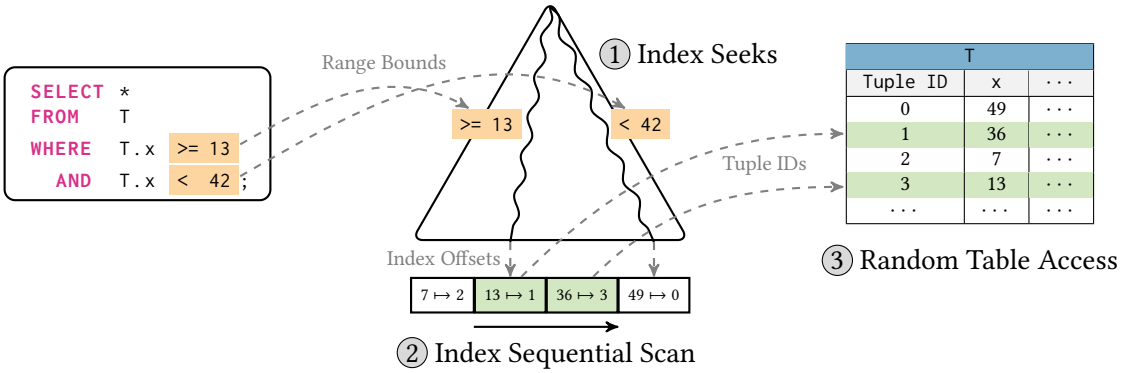


Figure 3.3: The individual operations involved in performing an index scan.

access method suitable for situations where the majority of tuples in a table are needed, i.e., either no filter conditions or filter conditions with very low selectivity are specified. However, a table scan can be inefficient for large tables if only a small subset of tuples is relevant, as it requires scanning through a significant amount of unnecessary data. Consequently, table scans are less optimal for queries that could benefit from more selective access methods.

Index Scan – An index scan leverages existing index structures to retrieve only the tuples that satisfy a specified filter predicate on the indexed attribute, effectively evaluating the filter predicate before loading the tuples. Unlike a table scan, which accesses tuples sequentially, an index scan typically loads tuples from the store using random accesses, as the qualifying tuples may not be stored consecutively in memory. These random accesses incur additional costs, making index scans most efficient for queries with high selectivity, where only a small subset of tuples is targeted, allowing for more efficient data retrieval compared to a table scan.

Although the index scan appears as a single operator in the QEP, it encompasses several distinct steps, detailed below and illustrated in Figure 3.3. We assume an index where entries can be sequentially scanned, such as the leaves of a B+Tree or index entries stored in a flat array. Further, we address the general case of a range predicate, with equality predicates being a specific instance where both bounds are identical. The steps involved are as follows:

① **Index Seeks**: Perform index seeks on the range bounds extracted from the SQL statement to locate the index offset for the first and last index entries within the queried range.

② **Index Sequential Scan**: Sequentially scan the qualifying index entries between the determined offsets to retrieve tuple IDs for the matching tuples. Depending on the implementation, the index may also point directly to the tuples.

③ **Random Table Access**: Load the corresponding tuples from the store using the retrieved tuple IDs through random table accesses.

An alternative approach would involve performing an index seek on the lower bound of the range only and then sequentially scanning the entries until they no longer match the range. However, this approach has significant disadvantages: we cannot determine the number of qualifying tuples without performing the entire index sequential scan, and during the scan, each entry must be checked to see if its corresponding key is within the desired range. Due to these drawbacks, we do not consider this variant of an index scan here.

Ultimately, both, table scan with filter and index scan, yield the same result: they identify and load the tuples that match the filter predicate. These tuples are then passed to subsequent operators in the QEP for further processing, ensuring the final query results are accurate and complete.

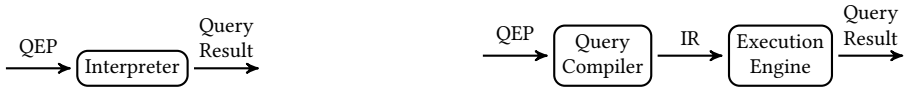
Example – Consider again the SQL query in Listing 3.1 along with the example data for table `T` shown on the right side of Figure 3.3. When processing the query with a table scan and a filter operator, as depicted in Figure 3.2b, each tuple is loaded from the table sequentially. After each tuple is loaded, the filter predicate specified in the `WHERE` clause of the query is evaluated. This process continues until all tuples have been processed, resulting in only the tuples with ID 1 and 3 being selected.

When processing the query with an index scan, as depicted in Figure 3.3, the lower bound (13, inclusive) and upper bound (42, exclusive) are first extracted from the filter condition. Next, index seeks are performed for these two bounds on the index on `T.x`, identifying the range of qualifying index entries from offset 1 (inclusive) to offset 3 (exclusive). The qualifying index entries are then sequentially scanned, and the corresponding tuples with tuple ID 1 and 3 are retrieved from the table via random access.

3.2.3 Query Execution Methods

After the optimizer has computed a QEP with specific operators and access methods, the query engine executes it to produce the query results. Various types of execution engines exist, with the primary distinction being between interpreting and compiling methods, both of which are explained below.

Interpretation – An interpreting query engine directly executes the operations of a QEP using generic operator implementations. The query engine consists of a single component, the interpreter, as illustrated in Figure 3.4a. This simple architecture makes interpreting query



(a) Interpreting query engine (b) Compiling query engine

Figure 3.4: Simplified architecture of interpreting and compiling query engines.

engines relatively easy to implement, debug, and maintain. Additionally, the interpreted execution provides a high degree of flexibility, allowing for dynamic optimizations based on runtime observations. However, interpretation introduces notable overhead due to frequent function calls for passing tuples between operators and dynamic type checking during expression evaluation. Historically, this overhead was insignificant in disk-based database systems, where the primary bottleneck was data retrieval from disk. With the rise of modern in-memory database systems, data retrieval is no longer the sole bottleneck, and computational efficiency has become crucial for achieving optimal performance.

One key inefficiency of interpretation stems from its traditional association with a tuple-at-a-time processing model, where individual tuples are processed sequentially. This approach incurs repeated costs for interpreting the same set of instructions for each tuple. To mitigate these inefficiencies, the vectorized processing model was developed [53, 7], allowing operators to process blocks, or vectors, of tuples rather than handling them individually. Vectorization reduces interpretation overhead by decreasing function calls, minimizing dynamic type checks, improving data locality, and enabling CPU parallelism.

It is important to note that interpretation and vectorization address different aspects of query execution: interpretation pertains to the execution strategy, while vectorization focuses on the granularity of data processing. While vectorization was originally introduced within the context interpretation, its principles can also be effectively applied to compilation.

Example – To illustrate the intricacies of interpreted execution, let us delve into the evaluation of a filter condition on a tuple. We focus on this particular aspect of the query, to highlight sources of interpretation overhead without delving into a specific execution model. This approach allows us to clearly see inefficiencies of interpreted execution, setting the stage for discussing the benefits of compiled execution.

The evaluation of a filter condition typically involves a function like `eval()` that is overloaded on all types of expressions, such as binary expressions, unary expressions, identifiers, and constants. In Listing 3.2, we present a code snippet showcasing the `eval()` function for binary expressions and identifiers. When evaluating the filter expression from the SQL query

in Listing 3.1 on a tuple, we initially invoke `eval()` on the expression and the tuple. The appropriate function is selected via dynamic dispatch by resolving the expression object's type at runtime.

Listing 3.2: Evaluating an expression in an interpreting query engine.

```
1  Value eval(BinaryExpression e, Tuple t) {
2      /* Determine expression operator. */
3      if (e.operator() == 'AND') {
4          /* Recursively evaluate left and right operands. */
5          Value l = eval(e.left(),  t);
6          Value r = eval(e.right(), t);
7          /* Return result of conjunction. */
8          return l and r;
9      }
10     ...
11 }
12
13 Value eval(Identifier i, Tuple t) {
14     /* Load value from tuple. */
15     return tuple.get_value(i);
16 }
```

In our scenario, the expression is a binary expression with an `AND` operator (Lines 1 and 3). Subsequently, `eval()` recursively processes the expression's left and right operands (Lines 5 and 6). This recursive process continues until it reaches a terminal, such as an identifier or a constant. For instance, when it encounters the identifier `x`, it retrieves the corresponding value from the tuple (Lines 13 to 15).

After evaluating both operands, `eval()` computes and returns the result of the `AND` operator (Line 8). This recursive traversal of the filter expression occurs for each tuple, introducing significant overhead due to frequent function calls and dynamic dispatches.

Compilation – A compiling query engine processes a QEP by generating optimized code for execution by the underlying hardware or runtime environment [55, 38]. The query engine consists of two main components, as illustrated in Figure 3.4b. The query compiler generates code from the QEP in the form of a low-level intermediate representation (IR). The execution engine then processes this generated code to compute the query result. Depending on the design of the query engine, the execution engine may further optimize and compile the IR into native machine code or directly interpret the IR.

Each system implementing query compilation employs a certain processing model that defines how the generated code handles data and control flow. In the following, we briefly explain the approach of Neumann to generating data-centric code, which is considered the foundation for

state-of-the-art execution engines [51]. Neumann proposes a processing model in which tuples are conceptually pushed from the leaves to the root of the QEP, where operators are joined into so-called pipelines, effectively blurring the boundaries of individual operators. Tuples are pushed until they reach an operator that requires materialization of intermediate results to be computed, for instance joins, grouping and aggregation, or sorting. This processing model has the decisive advantage that values are kept in CPU registers for as long as possible and memory is only accessed to load new tuples or materialize inevitable intermediate results.

Compilation addresses the main drawbacks of interpretation. Dynamic type checking is eliminated by generating code that is specialized for the attribute types. The number of function calls is reduced by replacing predicate evaluation with primitive data comparisons and employing a data-centric processing model that pushes tuples between operators instead of pulling them via function calls. However, implementing a compiling execution engine requires considerable engineering effort and is typically harder to debug compared to an interpreting query engine. In addition, optimization and compilation is costly and introduces a significant delay. While this optimization effort pays off for more complex analytical queries, interpretation is often preferable for simpler queries, as it starts the execution right away.

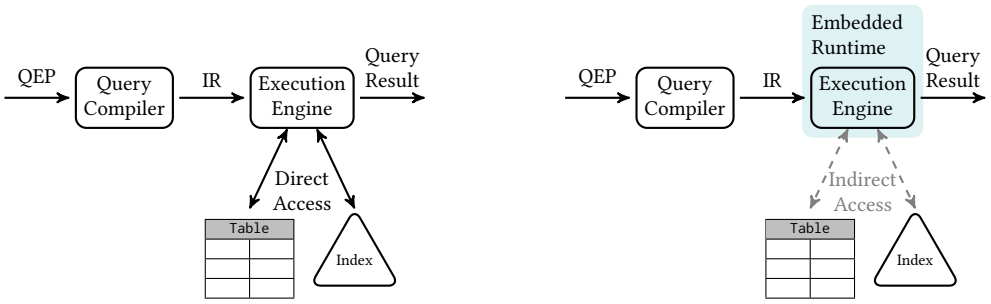
Listing 3.3: Generated code for the QEP in Figure 3.2b using the data-centric processing model.

```
1 /* Scan operator. */
2 for (Tuple t : T) {
3     /* Branching filter operator. */
4     if (t.x >= 13 and t.x < 42)
5         output(t)
6 }
```

Example – Listing 3.3 illustrates the code generated from the QEP in Figure 3.2b. The table scan operator on the bottom is transformed into a loop that iterates over all tuples in table `T`. The branching filter operator is realized using a straightforward if statement, with the filter predicate serving as the condition. Tuples that satisfy the condition are then returned. This example shows the flow of data, where tuples are pushed from the scan operator to the filter operator. In contrast to the interpreting approach for evaluating expressions depicted in Listing 3.2, the generated code eliminates the need for function calls, adopts a simpler structure, and implements the filter predicate using basic data comparisons.

3.2.4 Execution Environments

The execution environment is a crucial design aspect of a compiling query engine, defining how the execution engine interacts with the rest of the database system. These environments



(a) Integrated execution environment

(b) Isolated execution environment

Figure 3.5: Comparison of compiling query engines with integrated and isolated execution environments.

range from tightly integrated setups to isolated configurations, as illustrated in Figures 3.5a and Figure 3.5b. We explore these two primary approaches – integrated and isolated execution environments – in detail below.

Integrated Environment – In an integrated execution environment, the execution engine operates within the same process as the rest of the database system, enabling direct access to data sources such as tables and indexes. This setup allows the execution engine to directly invoke methods of these data structures, such as lookup methods for indexes, when executing queries, facilitating efficient communication and data transfer. Since these methods are compiled at system compile time, query compilation times are also reduced, as the execution engine can directly utilize them. However, the tightly coupled nature of this architecture often leads to monolithic designs, with reduced modularity and flexibility. This can make it more challenging to adapt or extend the system in certain ways. Examples of compiling query engines that operate within an integrated environment include HIQUE [38] and HyPer [51].

Isolated Environment – In contrast, an isolated execution environment separates the execution engine from the rest of the database by executing queries in an embedded runtime. This separation offers greater flexibility, modularity, and portability, but also introduces challenges for data access, as the execution engine cannot directly interact with tables and indexes in the database system. To address this, such systems typically adopt one of two approaches to facilitate communication between the execution engine and the database systems, which is referred to as the host in this context: (1) explicitly exposing host memory to the embedded runtime, or (2) using host calls to allow controlled interaction with other database components.

Exposing host memory allows the embedded runtime to directly access raw memory regions containing tables and indexes. This approach facilitates fast data access, but requires significant

engineering effort to ensure that the memory layout is correctly interpreted within the embedded runtime. As a result, the execution engine becomes tightly coupled with the host's memory layout. Any changes to the memory layout necessitate corresponding updates to the access logic on both the host and the embedded runtime, increasing maintenance complexity. Additionally, there are potential security concerns, as exposing raw memory can increase the risk of unintended data manipulation or access violations. Furthermore, depending on the embedded runtime, there may be restrictions on the amount and structure of the host memory that can be safely exposed to the embedded runtime, potentially limiting the flexibility of this approach.

Host calls allow the embedded runtime to communicate with the database system through a predefined interface provided by the host. This approach offers better abstraction by decoupling the execution engine from the memory layout of specific database components, in contrast to the approach of exposing host memory. Instead, the execution engine interacts with the host interface, which provides access to tables and indexes. Internally, host calls may invoke methods on these data structures or utilize other abstractions provided by the database system, streamlining access logic. The results of these host calls are communicated back to the embedded runtime either directly as return values or by writing them to a shared memory region accessible by the embedded runtime. Since the host interface is compiled at system compile time, the use of host calls reduces query compilation times by allowing the execution engine to utilize these precompiled interfaces, much like in an integrated environment. However, host calls introduce significant performance overhead due to context switches between the embedded runtime and the host. This overhead is further compounded by marshalling and unmarshalling of call parameters, particularly in scenarios involving complex and frequent interactions.

An example of a compiling query engine with an execution engine operating in an isolated environment is mu*t*able, which employs a combination of exposed host memory and host calls. The mu*t*able system is the basis for our experiments in Section 3.6.

3.3 Query Compilation With Partial Execution

Traditional query engines face a trade-off between compilation, which produces highly optimized code but introduces compilation overhead, and interpretation, which allows for immediate execution but suffers from interpretation overheads. A key advantage of interpretation is its ability to dynamically adjust the execution based on runtime observations. For instance, an interpreting query engine can monitor the selectivity of individual clauses within

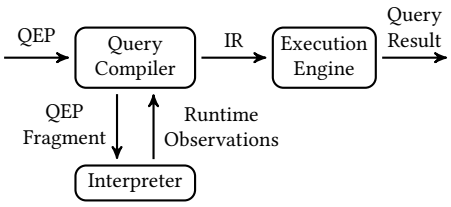


Figure 3.6: Architecture of a compiling query engine with partial execution during query compilation. The query compiler extracts a QEP fragment for interpretation and integrates runtime observations into the generated IR code.

a conjunctive predicate and dynamically reorder them to evaluate the most selective clause first. Our approach is a step towards bringing similar flexibility to compiled execution by augmenting the compilation process with runtime observations, such as intermediate results or statistical properties of the processed data. In the following sections, we introduce the general architecture of our query engine (Section 3.3.1) and then explain why index scans are a suitable candidate for applying this technique (Section 3.3.2).

3.3.1 General Architecture

Our architecture enhances IR generation by incorporating runtime observations. To achieve this, we introduce a novel query engine design, shown in Figure 3.6, that allows the query compiler to partially execute QEPs – or specific QEP fragments – during the compilation phase, before the actual execution phase begins. This partial execution provides valuable insights, such as statistical properties of the processed data or intermediate results, which the query compiler then uses to enhance and optimize the generated IR code.

In this context, a QEP fragment is a subset of a QEP that consist of a data source, such as a (partial) table scan or an index access, followed by any number of additional operators. Unlike a full subtree of a QEP, the operators in a fragment do not necessarily represent a complete execution step. Instead, operators may only execute a subset of their operations to gather insights into specific execution behaviors or process a restricted portion of the tuples to obtain empirical distribution information from a data sample.

When executing a query, the query compiler first receives the QEP and identifies relevant QEP fragments for partial execution. These fragments are then sent to an interpreter, which executes them and returns runtime observations along with intermediate results. Using these insights, the query compiler incorporates the gathered information into the IR code generation process.

Depending on the size of the QEP fragments that are interpreted, this approach enables us to retain the performance benefits of full compilation while leveraging some runtime observations to generate more informed and efficient IR code. This approach is particularly advantageous in scenarios where precise knowledge of intermediate results can help the execution engine further optimize the generated IR code. Additionally, the architecture allows for a more fine-grained control over interactions with data sources like tables or indexes. Data sources may either be accessed during compilation using interpreted execution or at runtime from within the generated IR code using compiled execution. This flexibility is especially beneficial in environments where accessing data sources at runtime is costly or impractical, such as within an isolated execution environment, as it enables shifting these accesses to compile time.

The idea of executing parts of a program during compilation to gather insights for optimizations is not new and has been explored in various compilation techniques. Concepts such as partial evaluation [17] leverage partial execution to specialize code based on known inputs. Our architecture extends this general principle to query processing by enabling the query compiler to execute QEP fragments at compile time. Unlike, traditional query optimization, which relies on static cost models or limited sampling, this approach provides a flexible and systematic way to incorporate empirical data into the compilation process, bridging the gap between compile-time optimization and runtime adaptability.

Example – Consider a query with a conjunctive predicate consisting of multiple clauses, where each clause filters a portion of the data. The query compiler can extract a QEP fragment that represents the evaluation of these clauses, and during partial execution, it can make a runtime observation based on a sample of the data to estimate the selectivity of each individual filter clause. Based on these observations, the query compiler can reorder the clauses in the conjunctive predicate so that in the generated IR code, the most selective clauses are evaluated first. This reordered evaluation allows the effective use of early stopping: for each tuple, if a clause evaluates to false, the evaluation of the remaining clauses in the conjunctive predicate can be skipped. Since the reordering improves the chances of filtering non-qualifying tuples early, this avoids unnecessary work and has the potential to significantly speed up the evaluation of the conjunctive predicate. Additionally, intermediate results containing tuples already identified as qualifying can be embedded directly into the generated IR code, eliminating redundant computations.

Another example of how runtime observations improve query execution is when the exact number of intermediate results is known. Consider a scenario where a QEP fragment consists of a table scan or an index scan that produces a set of qualifying tuples. With precise knowledge

of the number of qualifying tuples, the query compiler can apply optimizations like fully unrolling loops that would otherwise be impossible with only static analysis. Additionally, by determining the iteration range, unnecessary bound checks can be omitted, further reducing execution time.

3.3.2 Index Scans as a Suitable Candidate

Index scans are particularly well-suited for integration with a query engine that supports partial execution during compilation. Several characteristics make them an ideal candidate for exploring the potential of this approach.

First, as outlined in Section 3.2.2, index scans naturally decompose into multiple distinct operations. This decomposition provides a variety of potential QEP fragments that can be selectively interpreted during compilation.

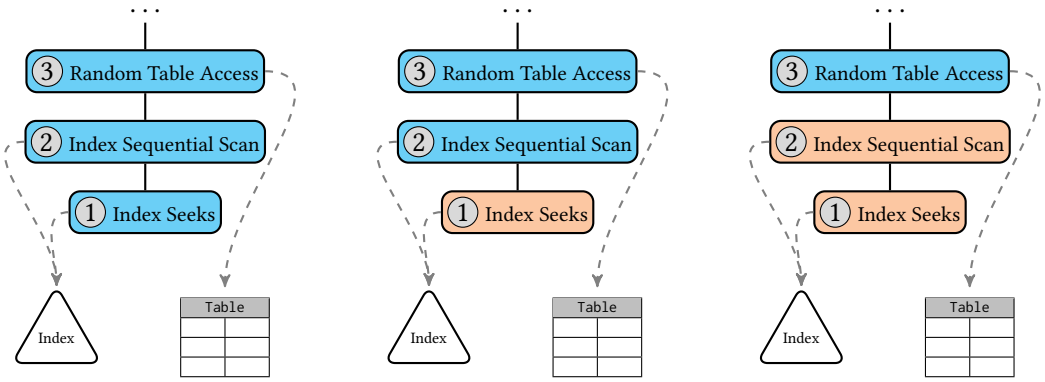
Second, as index scans occur at the leaves of a QEP, the corresponding QEP fragments typically consist only of operations directly involved in the index scan. These QEP fragments can be executed quickly in the interpreter, allowing us to efficiently gather runtime observations without introducing significant overhead.

Third, index access patters typically do not suffer from the interpretation overhead seen in other query operators. Since indexes are usually fully typed and traversing an index involves only a fixed number of function calls, the overhead remains the same whether these calls are made from an interpreted execution or from the generated code in a compiled execution. This ensures that interpreting index-related QEP fragments incurs minimal additional cost while still providing valuable insights for IR code generation.

These properties make index scans a natural candidate for exploring different ways of integrating index access into the compilation process. Next, we introduce three distinct index access strategies, which define how and to what extent interaction with an index is shifted from the query execution phase into the compilation phase.

3.4 Index Access Strategies

This section introduces three distinct strategies for accessing indexes as part of an index scan. Building on the architecture discussed in Section 3.3.1, these strategies define the extent to which index operations are preformed during compilation rather than at runtime. We present the compiled (Section 3.4.1), interpreted (Section 3.4.2), and hybrid index access



(a) Compiled index access (b) Hybrid index access (c) Interpreted index access

Figure 3.7: Overview of three index access strategies, with colors indicating the component responsible and the execution method for each operation in an index scan: interpreted execution in the query compiler (orange) and compiled execution in the execution engine (cyan).

strategies (Section 3.6.5), and conclude with a discussion of alternative approaches that were excluded due to misalignment with our design goals (Section 3.4.4).

3.4.1 Compiled Index Access Strategy

In the compiled index access strategy (illustrated in Figure 3.7a), the query compiler generates IR code for the entire QEP without extracting any QEP fragments for partial execution during compilation. This means that index access operations, such as index seeks and index sequential scan, and random table access are fully integrated into the generated IR code and executed at runtime. Since all interactions with the index occur within the compiled query code, this approach avoids reliance on runtime observations or intermediate results.

This strategy aligns with how most traditional compiling query engines operate, prioritizing execution performance by completely eliminating interpretation overhead. However, the effectiveness can vary depending on the system architecture. A key factor is whether the generated IR code runs in the same integrated environment as the database system or in an isolated environment.

Integrated Environment – When the compiled index access strategy is applied in an integrated environment, the generated IR code can directly interact with data members and invoke member functions of the database system. This allows seamless access to index structures without additional abstraction layers. As a result, both index seeks and index

sequential scan are executed efficiently within the execution engine. Since member functions are already part of the database system and compiled at system compile time, the execution engine can utilize them as precompiled routines when executing the IR code. This reduces the need for regenerating index access logic, leading to faster compilation times.

The combination of direct access to index structures and reduced compilation overhead makes this approach the default in most compiling database systems that operate in an integrated environment. By leveraging precompiled routines and avoiding unnecessary indirections, these systems achieve both high performance and low query compilation latency.

Isolated Environment – When the compiled index access strategy is applied in an isolated environment, the generated IR code cannot directly access index structures or invoke member functions. To circumvent this restriction, one approach is for the host to provide an interface for accessing the index. IR code generation in this case mirrors the integrated environment, replacing direct member function calls with calls to the host-provided interface. While this avoids regenerating access logic and benefits compilation time, the execution overhead from context switches and marshalling remains. Batching multiple index operations into a single call can reduce this overhead but comes at the cost of materializing larger intermediate results.

Another approach is for the host to expose raw memory, allowing the execution engine to bypass host calls entirely. However, this requires replicating the index access logic in the target IR language. This eliminates the need for host calls and their associated overhead but requires significant engineering effort. While parts of this logic may be precompiled into libraries, ensuring compatibility with the host's memory layout remains a challenge.

Listing 3.4: Generated code for the QEP in Figure 3.2c using the compiled index access strategy.

```
1  /* Seek index on T.x to determine entry offsets. */
2  Offset lo = index.lower_bound(13)
3  Offset hi = index.lower_bound(42)
4  /* Sequentially scan index entries. */
5  while (lo < hi) {
6      Entry e = index.get_entry(lo)
7      /* Randomly access tuple in table T. */
8      Tuple t = store.load(T, e.tuple_id)
9      output(t)
10     lo++
11 }
```

Example – Listing 3.4 demonstrates the code generated from the QEP in Figure 3.2c using the compiled index access strategy. In an integrated environment, function calls to `index` and `store` are direct member function invocations. In contrast, in an isolated environment, these

calls are made through the host interface. Note that we omit the code for the exposed host memory case, as it would depend on the specific index type and memory layout of the host.

The code begin by performing two index seeks to determine the range of qualifying index entries (Lines 2 and 3). Subsequently, these entries are scanned sequentially (Lines 5 and 6), and the corresponding tuples are retrieved from the store via random table access (Line 8).

Summary – The compiled index access strategy aims to minimize interpretation overhead by generating specialized IR code for the entire QEP, including all interactions with indexes. In an integrated environment, the IR code can directly access data members and invoke member functions. In an isolated environment, interaction with the index at runtime is facilitated whether through potentially costly host calls or by operating on raw memory exposed by the host.

3.4.2 Interpreted Index Access Strategy

In the interpreted index access strategy (illustrated in Figure 3.7c), the query compiler extracts a QEP fragment that encapsulates all index interactions. This fragment includes index seeks to determine offsets into the index entries, followed by an index sequential scan that retrieves the tuple IDs of qualifying tuples. Instead of generating IR code for these operations, the query compiler sends the fragment to the interpreter, which executes it during query compilation and produces tuple IDs as the intermediate results. These results are then materialized and integrated into the generated IR code. The materialization can be implemented in two ways: inlining tuple IDs as constants directly in the IR code or by writing them to memory and generating IR code that reads and processes them at runtime.

Materializing qualifying tuple IDs opens opportunities for the execution engine to apply additional optimizations. For instance, if the set of qualifying tuple IDs is known at query compile time, the execution engine can fully unroll loops over these IDs, eliminating loop overhead and reducing branching. Additionally, memory access patterns for loading tuples from the store can be optimized by reordering accesses to improve cache locality and prefetching data to minimize latency. Nonetheless, these optimizations come at the cost of having to materialize the qualifying tuple IDs, which increases the size of the compiled query. As a result, the size of the compiled query not only depends on algorithmic choices but also on the number of qualifying tuples, potentially impacting the performance on resource-restricted devices – an issue not present in the compiled index access strategy.

Compared to the compiled index access strategy, this approach is much less influenced by the execution environment. By shifting all index interactions to query compilation, it eliminates the need for index access during execution. However, subtle differences remain, particularly in how intermediate results are stored and accessed at runtime

Integrated Environment – In an integrated environment, execution is straightforward. If the tuple IDs are inlined as constants in the IR code, the execution engine directly loads the corresponding tuples via random table access. If tuple IDs are materialized in memory, the execution engine, operating within the same environment as the rest of the database system, can directly access this memory to retrieve the tuple IDs at runtime and load the tuples accordingly.

Isolated Environment – An isolated environment benefits from the interpreted index access strategy entirely eliminating index interactions at runtime by shifting them to the query compiler, which operates within the same environment as the rest of the database system. This avoids both costly host calls and the need to replicate access logic for exposed host memory, depending on the implementation.

If tuple IDs are inlined as constants, the execution behaves similarly to the integrated environment. However, if tuple IDs are materialized in memory, the execution engine must have access to that memory. This requires either allocating space within the embedded runtime or exposing host memory to the embedded runtime.

Listing 3.5: Generated code for the QEP in Figure 3.2c using the interpreted index access strategy.

```
1 /* Tuple IDs of qualifying tuples are inlined. */
2 List tuple_ids = [3, 1]
3 for (tuple_id : tuple_ids) {
4     /* Randomly access tuple in table T. */
5     Tuple t = store.load(T, tuple_id)
6     output(t)
7 }
```

Example – Listing 3.5 demonstrates the code generated from the QEP in Figure 3.2c using the interpreted index access strategy with inlined intermediate results. Index seeks and index sequential scan are already performed at compile time, and the resulting tuple IDs of the qualifying tuples are inlined directly into the generated code (Line 2). The code then iterates over these tuple IDs in a loop (Line 3), retrieving the corresponding tuples from the store via random table access (Line 5). As there is no interaction with the index at query runtime, the

code is identical whether it is executed in an integrated or isolated environment. In the isolated environment, however, calls to the `store` are made through host calls.

Summary – The interpreted index access strategy extracts a QEP fragment that includes index seeks and index sequential scan, effectively eliminating index interaction at runtime. The tuple IDs resulting from executing the fragment are materialized to communicate them to the execution engine. While integrating the tuple IDs into the generated IR code enables the execution engine to perform additional optimizations, this approach may also increase memory consumption and code size of the compiled query.

3.4.3 Hybrid Index Access Strategy

The hybrid index access strategy (illustrated in Figure 3.7b) combines both interpretation and compilation for the index access. The query compiler extracts a QEP fragment containing only the index seeks, which it sends to the interpreter for execution during compilation. The resulting offsets into the index are integrated into the generated IR code, again either by inlining them as constant or by materializing them in memory. Based on these offsets, the index sequential scan is performed at runtime.

Materializing the index offsets opens up several optimization opportunities, as it provides the execution engine with knowledge of the exact number of qualifying tuples the index sequential scan and random table accesses will produce. This foresight allows the execution engine to optimize memory allocation and loop structures, such as preallocating memory for the qualifying tuple IDs or unrolling loops based on the known number of results. However, if the index offsets are materialized in memory the number of iterations is not explicitly present in the generated IR code and the distribution of qualifying tuples remains unknown at compile time, optimization potential is limited and certain optimizations, such as loop unrolling or specialized data access patterns, cannot be applied.

Compared to the interpreted index access strategy, the hybrid index access strategy only materializes the index offsets, making the memory consumption of the compiled plan independent of the number of qualifying tuples. However, like the compiled index access strategy, the execution of the hybrid index access strategy is not independent of the execution environment, as there is interaction with the index at query runtime.

Integrated Environment – In an integrated environment, execution is again straightforward. If the offsets are inlined as constants in the IR code, the execution engine performs a fixed number of loop iterations to sequentially scan the index entries and retrieve the corresponding

tuple IDs. When materialized in memory, the execution engine can access the offsets directly since it operates within the same environment as the rest of the database system. Based on the offsets, the execution engine then loops over the index entries and retrieves the tuple IDs in the same manner.

Isolated Environment – In an isolated environment, the hybrid index access strategy benefits from the partial elimination of index interactions at runtime. If host calls are used, the hybrid index access strategy eliminates the host calls for index seeks by resolving them during the query compilation. Similar to the compiled index access strategy, the remaining host calls, needed for the index sequential scan, can be batched to retrieve multiple tuple IDs at once. This reduces the overhead due to host calls compared to the compiled index access strategy but still incurs more than the interpreted index access strategy. If exposed host memory is used instead, only the logic for sequentially scanning index entries needs to be replicated in the target language, leading to more duplication than in the compiled strategy but less than in the interpreted index access strategy. As with the interpreted index access strategy, when materializing index offsets in memory, care must be taken to ensure the embedded runtime can access this memory at runtime to retrieve the index offsets.

Listing 3.6: Generated code for the QEP in Figure 3.2c using the hybrid index access strategy.

```
1  /* Index offsets are inlined. */
2  Offset lo = 1
3  Offset hi = 3
4  /* Sequentially scan index entries. */
5  while (lo < hi) {
6      Entry e = index.get_entry(lo)
7      /* Randomly access tuple in table T. */
8      Tuple t = store.load(T, e.tuple_id)
9      output(t)
10     lo++
11 }
```

Example – Listing 3.6 demonstrates the code generated from the QEP in Figure 3.2c using the hybrid index access strategy. In this example, index seeks are executed during query compilation, and the resulting index offsets are inlined in the generated code, defining the range of qualifying index entries (Line 2 and 3). At runtime, the execution engine performs an index sequential scan over this range (Lines 5 and 6) and retrieves the corresponding tuples from the store via random table access (Line 8). In an integrated environment, calls to `index` and `store` are direct member function invocations. In contrast, in an isolated environment, these calls are made through the host interface. Again, we omit the code for the exposed host memory case, as it would depend on the specific index type and memory layout of the host.

The generated code closely resembles that of the compiled index access strategy, with the key difference being that the range of qualifying entries is known at compile time, allowing for additional compiler optimizations.

Summary – The hybrid index access strategy combines interpretation and compilation for index access, executing index seeks during query compilation while performing the index sequential scan at runtime. By determining the exact number of qualifying tuples at compile time, it enables more optimizations than the compiled index access strategy but fewer than the interpreted index access strategy. The approach introduces more interpretation overhead than the compiled index access strategy but less than the interpreted one. Overall, the hybrid index access strategy balances the optimization potential of the generated IR code with the interpretation overhead of executing larger QEP fragments and cost of materializing larger intermediate results.

3.4.4 Discussion

Two potential points of critique regarding the strategies are the exclusion of random table accesses from interpretation during query compilation and the decision to statically switch from interpretation to compilation. In the following, we will reason why these decisions were made and discuss implications of each choice.

Excluding Random Table Access – Consider again Figure 3.7, which compares the three strategies. Each of these strategies either employs an interpreting approach for index access during compilation, a compiled approach for index access at runtime, or a combination of both. However, all strategies consistently use a compiled approach for table access at runtime. There are two key reasons why interpretation was not considered for table access.

First, our work is focused on index interaction and explores the potential of shifting those interactions from runtime to the compilation phase. Second, interpreting table access is typically inefficient, as it introduces significant overhead due to the need to process both the data layout and the table schema. Additionally, interpreted table access would require materializing the relevant attributes of all qualifying tuples instead of passing tuple IDs or offsets into the index. This would increase both memory consumption and I/O costs, as the tuples would need to be loaded and materialized during compilation and then processed again at runtime. Furthermore, materializing tuples disrupts the pipelined execution model, where tuples are typically passed directly between operations as they are produced, introducing further inefficiencies. Therefore, we decided to not include a strategy that interprets the random table access.

Fixed Transition Point – The presented strategies either transition from interpreted execution in the query compiler to compiled execution in the execution engine once (interpreted and hybrid index access strategy), or do not use interpretation at all (compiled index access strategy). Notably, none of the strategies involve transitioning back and forth between interpreted and compiled execution. Below, we outline two potential implementations of such a strategy and explain why we believe them to be inferior in terms of performance compared to the presented strategies.

The first implementation would alternate between interpretation and a compilation phases. Such an approach has two major disadvantages. Firstly, to communicate intermediate results between two consecutive phases, these results must be fully materialized increasing memory consumption and disrupting pipelined execution. Secondly, generated IR code must be optimized and compiled whenever switching from interpretation to compilation, inducing significant overhead. These factors likely negate any potential benefits from optimizations or avoided host calls that this strategy may offer.

The second implementation involves using interpretation by calling an interpreter from the generated code. Although this approach only requires a single compilation step, it limits the opportunities of compiler optimizations. Furthermore, if executed within an isolated environment, the interpreter must be invoked via host calls, incurring not only interpretation overhead but also overhead from context switches. Therefore, we conclude that this strategy does not provide advantages over the strategies presented.

3.5 Implementation Details

In this section, we delve into the specifics of the strategy implementations used in our experimental evaluation. We start by outlining the architecture of mu*t*able's query engine, which serves as the foundation of our implementation (Section 3.5.1). Next, we provide a comprehensive overview of the different strategy variants we have implemented (Section 3.5.2). Finally, we detail the integration of two indexes into mu*t*able for use in index scans (Section 3.5.3).

3.5.1 System

We implemented the strategies in mu*t*able [26], a main-memory database system designed for prototyping research ideas, written in C++. Its modular design and clear separation of concerns allow researchers to replace specific components with minimal effort and to accurately measure

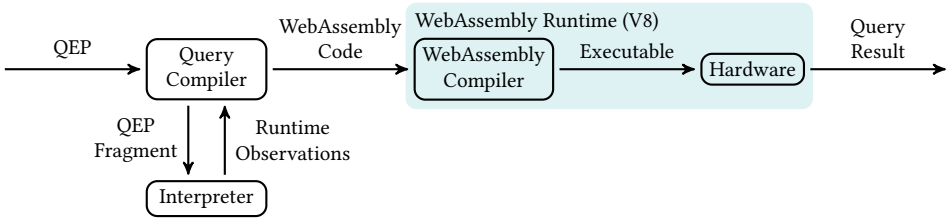


Figure 3.8: Simplified architectural overview of mutable's compiling query engine.

the impact of each change. mutable is a compiling database system that generates data-centric code, as introduced in Section 3.2.3. Since our work focuses on different index access strategies for index scans, we turn our attention to mutable's query engine, a simplified architectural overview of which is shown in Figure 3.8.

Query Compilation – mutable compiles QEPs into WebAssembly (WASM), a low-level code format designed for efficient execution and compact representation [68]. Each QEP is translated into a *module*, a static representation of a WebAssembly program. A module serves as a logical unit consisting of functions and global variables with its own storage. This storage, known as *linear memory*, is a contiguous array of uninterpreted bytes. The query compiler first dissects a given QEP into pipelines and subsequently compiles each pipeline into a separate function. Afterward, a `main()` function is created to coordinate the execution of these pipelines. The generated WebAssembly code assumes that the linear memory holds all required tables. Additionally, the linear memory serves as heap for materializing intermediate and final results.

We extended mutable's query compiler to integrate an interpreter, enabling it to partially execute QEPs during compilation. Currently, there is no clear separation or well-defined interface between the query compiler and the interpreter. Instead, while traversing the QEP to generate WebAssembly code, the query compiler may invoke interpreter routines as needed, such as evaluating predicates on tuples or extracting bounds from conjunctive range predicates. We plan to introduce a cleaner interface along with a dedicated data structure to represent QEP fragments in the future.

WebAssembly Compilation and Execution – mutable delegates the compilation and execution of WebAssembly code to V8 [21], Google's JavaScript and WebAssembly engine. V8 is embedded in mutable and executes the compiled queries in an isolated environment. At the time of writing, V8 does not support directly importing memory from the host into a module, despite WebAssembly itself not imposing such a restriction [68]. To allow the modules access to tables residing in the host, we use a patched version of V8 and a technique called

Table 3.1: Index access strategy variants applicable to an isolated execution environment.

Strategy	Runtime index access	Intermediate result materialization	Implementation status
Compiled index access	Host calls	–	✓
	Exposed host memory		✗
Interpreted index access	–	Code inlining	✓
		Memory	✓
Hybrid index access	Host calls	Code inlining	✓
	Host calls	Memory	✓
	Exposed host memory	Code inlining	✗
	Exposed host memory	Memory	✗

rewiring [56]. The patched version of V8 lets us replace the linear memory of a compiled module with arbitrary virtual memory. Rewiring enables us to map the physical memory of several tables to contiguous virtual addresses.

During the execution, we first let V8 compile the module, resulting in an executable. We then replace the module's memory with a virtual address space containing all required tables and start the execution by invoking the module's `main()` function. The module writes query results in its linear memory, which the host reads after completion. Using V8 offers several benefits. Most notably, it supports dynamic tier-up, a form of adaptive execution where the WebAssembly module is initially compiled without any optimizations to minimize latency, while the code is optimized and recompiled in the background. This allows V8 to seamlessly transition to the optimized execution at a later time. Additionally, V8 automatically caches compiled modules, preventing the recompilation of previously compiled modules. For further details on mu*t*able's query engine, consider the excellent work by my colleagues Haffner and Dittrich [25].

3.5.2 Index Access Strategies

As previously detailed in Section 3.4, each strategy offers multiple implementation variants primarily distinguished by how indexes are accessed at runtime (via host calls or exposed host memory) and how intermediate results, such as tuple IDs and index offsets, are handled (materialized in memory or inlined in code). Table 3.1 provides an overview of these variants applicable to mu*t*able's isolated execution environment.

We also indicate the implementation status of each variant. Currently, we have not implemented any variant that uses exposed host memory for accessing the index at runtime. This limitation is due to current restrictions in V8, which does not support importing host memory directly into a module at the time of writing. To circumvent this restriction, rewiring [56] could be employed, similar to how table accesses are implemented. However, such an approach would require both substantial modifications to mutable's query engine and replication of the index access logic in WebAssembly. Due to time constraints, we have been unable to implement index access via exposed host memory. However, we intend to complete the implementation of these variants as part of future work. Detailed implementation specifics for each strategy are provided below.

Compiled Index Access Strategy – Our implementation of the compiled index access strategy makes use of two host functions that are called to obtain the qualifying tuple IDs. The first host function performs an index seek for a given key and returns the offset of the corresponding index entry to the WebAssembly module. This function is called twice, once for the left and once for the right bound according to the filter predicate. From those offsets, we can compute the number of qualifying tuples.

The second host function returns the tuple ID stored in the index entry at a given offset. This function performs the index sequential scan by being invoked on each offset within the range from the first to the last offset obtained from the previous host function. Additionally, this function accepts batches of offsets to scan multiple entries in a single host call. In contrast to the host function performing the index seek, this function does not return the resulting tuple ID by value but instead writes it to a provided memory address in the module's linear memory. The memory where the host function writes the tuple ID is allocated at runtime. This approach enables us to easily extend the host function to support batching by writing multiple tuple IDs per host call. We never allocate more memory than required by choosing the minimum of specified batch size and number of qualifying tuples.

Interpreted Index Access Strategy – The interpreted index access strategy calls member functions of the index to obtain the index entries matching the filter predicate. While iterating over these entries, we materialize the qualifying tuple IDs either by inlining them in the generated code or by writing them to memory. When inlining the tuple IDs, we define a function in the WebAssembly module which, given a tuple ID, loads the respective tuple from the store and further processes it according to the pipeline. The tuple IDs are then inlined by generating a function call for every qualifying tuple ID. This approach avoids replicating the

code of the pipeline for each qualifying tuple and eliminates the loop over the qualifying tuple IDs, resulting in overall simpler code.

When writing the tuple IDs to memory, we initially seek the first and last offset to determine the number of qualifying tuples. Then, we allocate sufficient memory for all qualifying tuple IDs in the virtual address space that later replaces the module's linear memory. Next, we write the tuple IDs to that memory. Finally, we generate code that iterates over this memory and reads the tuple IDs, loads the corresponding tuples, and further processes them according to the pipeline.

Hybrid Index Access Strategy – The hybrid index access strategy combines the implementation principles of both the compiled and the interpreted index access strategy. During query compilation, we perform two index seeks to obtain the first and last offset. These offsets are again either materialized by inlining or by writing them to memory. When offsets are inlined, they are written into the code as constants and used for host calls to iterate over the qualifying index entries. When offsets are written to memory, we allocate memory for the two offsets in the virtual address space that becomes the module's memory and write the offsets there. The generated code then reads the two offsets and stores them in variables. In both cases, the hybrid index access strategy uses the same host function as the compiled index access strategy to perform the index sequential scan. Therefore, it also supports batching and ensures that no more memory is allocated than necessary.

All strategies extract the bounds from the filter condition by traversing the corresponding expression once during query compilation. Similarly, all strategies resolve the indexed attribute's type to produce fully specialized code once during query compilation. Both the host functions and the member functions of the index, whether invoked within the host functions or as part of interpretation, are specialized for that attribute's type, thereby eliminating any unnecessary interpretation overhead.

3.5.3 Indexes

While our evaluation focuses on comparing the three strategies, we also analyze each strategy using two different types of indexes that we implemented in mu*t*able.

Sorted Array – The first index is a contiguous array of sorted index entries, where each entry consists of a key and a tuple ID. An index seek is carried out by performing binary search on the entries. An index sequential scan iterates over a specific range of index entries.

Recursive Model Index – The second index is the recursive model index (RMI) [37], a read-optimized learned index, previously introduced in Section 2.2. The RMI is based on the observation that the offset of an entry in a sorted array can be computed using the cumulative distribution function (CDF) of the entries' keys. The RMI aims to approximate this CDF with a hierarchical model. Since the RMI only approximates the CDF, any prediction errors must be corrected by performing a local search around the predicted offset.

The foundation of the RMI is again a sorted array, with a hierarchical model on top for faster retrieval of entries. An index seek uses the hierarchical model to predict an offset in the sorted array, followed by a local search for the desired entry. An index sequential scan is conducted similarly to the sorted array by iterating over a range of index entries in the sorted array. The RMI in mutable is a simplified version of our open-source implementation [43], utilizing a fixed combination of model types for the hierarchical model: a linear spline model for the first layer and linear regression models for the second layer. We use exponential search to correct potential prediction errors and choose a model-to-entry ratio of 0.01 for the second layer, which is a sane choice for achieving good lookup performance [44].

To minimize any interpretation overhead, both indexes are specialized for the key type. Consequently, the type only needs to be resolved once when processing a query, regardless of whether interpretation or compilation is used. Both indexes are implemented as secondary indexes and stored separately from the associated table. This implies that an index scan needs to access both the index and the table. Further, both indexes map keys to tuple IDs, as mutable uses tuple IDs in its interface to the store for loading single tuples via random access. Tuple IDs are unsigned 32-bit integers.

3.6 Experimental Evaluation

In this section, we present our experimental evaluation of the three index access strategies. We begin by outlining our experimental setup (Section 3.6.1). The evaluation is divided into four parts: first, we compare the strategies in terms of execution time (Section 3.6.2); second, we investigate when to use which variant of each strategy (Section 3.6.3, 3.6.4, and 3.6.5); third, we examine how the choice of index impacts overall execution time (Section 3.6.6); and fourth, we explore whether the strategies benefit from caching compiled plans of previous queries (Section 3.6.7).

3.6.1 Experimental Setup

Machine – The experiments are conducted on a Linux machine with an Intel® Xeon® E5-2620 v4 CPU (2.10 GHz, 512 KiB L1, 4 MiB L2, 20 MiB L3) and 4x8 GiB DDR4 RAM. Our code is compiled with `clang-16.0.6` at optimization level `-O2` and executed single-threaded.

System – We implemented the strategies in mu*t*able, as described in Section 3.5.2. We set up mu*t*able to refrain from printing query results to the console, while ensuring that the query results are materialized in memory to prevent loading instructions from being optimized away. Unless explicitly stated otherwise, V8, responsible for compiling and executing the generated code, is configured with both adaptive execution and caching of compiled modules turned off to obtain stable and traceable measurements.

Indexes – For evaluating the index scan strategies, we implemented two types of secondary indexes, as described in Section 3.5.3: a sorted array and an RMI. The sorted array serves as the baseline index and is used in all experiments, unless explicitly specified otherwise.

Datasets – Our evaluation is based on four simple datasets, each comprising two columns: an `id` column and a `data` column. The `id` column consists of 32-bit signed integers, sequentially incremented starting from zero. The `data` column consists of unique and unordered numeric values, generated uniformly at random. Each dataset uses a different data type for the `data` column, either 32-bit or 64-bit floating-point numbers or 32-bit or 64-bit signed integers. The datasets consist of 1M tuples and are stored in row layout to simplify address calculations for random accesses. Indexes are constructed on the `data` column.

Workload – For each of the four datasets, we generate seven queries with filter predicates varying in selectivity between 10^0 (all 1M tuples qualify) and 10^{-6} (a single tuple qualifies). We strive to maintain simplicity in the generated queries to isolate the performance impact of the strategies, ensuring that it is not overshadowed by other operators. The queries have the following format:

```
1 SELECT T.id, T.data
2 FROM T
3 WHERE T.data >= x AND T.data <= y;
```

Constants `x` and `y` in the `WHERE` clause are chosen to match the target selectivity.

Measurements – We focus exclusively on times related to query execution and neglect query optimization time, as the optimization effort remains identical across all queries and strategies within our workload. Measurements are defined as follows:

- **Machine code compilation time** refers to the time V8 takes to compile the generated WebAssembly code into executable machine code.
- **Execution time** includes the time required for our query compiler to generate WebAssembly code for a given QEP, potentially involving partial execution of QEP fragments, plus the time V8 takes to compile and execute that code.

We execute experiments five times and report the median of the measured times, respectively.

Baseline – As a reference point, we benchmark the strategies against a sequential table scan followed by a branching filter operator, as explained in Section 3.2.2.

3.6.2 Comparing the Index Access Strategies

The initial experiment compares the strategies in terms of execution time while using the optimal variant of each strategy. To achieve this, we first ran the experiment with all implemented strategy variants listed in Table 3.1, varying the batch size for the hybrid and compiled index access strategies. We then determined the optimal execution time for each strategy on each dataset and query selectivity. This implies that the same strategy is not necessarily configured equally across datasets and selectivities. Subsequent experiments explore each strategy in more detail, comparing the different variants of a strategy. Figure 3.9 presents a comparison of execution times among the three strategies and the baseline.

For each strategy, execution times increase when selectivity decreases. This increase is expected, as a lower selectivity means that more index entries must be scanned, and more tuples have to be loaded from the store and materialized as query result. In our experiment, index scans outperform the table scan for selectivities higher than 10^{-1} (100k qualifying tuples), while the table scan performs better at lower selectivities. Additionally, we observe slightly higher execution times for 64-bit data types than for 32-bit data types, particularly visible at lower selectivities. This is due to higher costs associated with materializing the query results.

Overall, the three strategies exhibit similar performance. The interpreted index access strategy outperforms the other two strategies for selectivities of 10^{-1} (100k qualifying tuples) and higher. At these selectivities, the avoidance of host calls associated with the interpreted strategy outweighs the additional effort of materializing intermediate results, as the tuple IDs remain relatively small. Additionally, the qualifying tuple IDs materialized during query compilation still fit into the L1 cache, suggesting that the strategy benefits from at least partial caching of the tuple IDs at runtime. For selectivities lower than 10^{-1}, the hybrid and compiled index access strategies perform slightly better than the interpreted index access strategy. Both

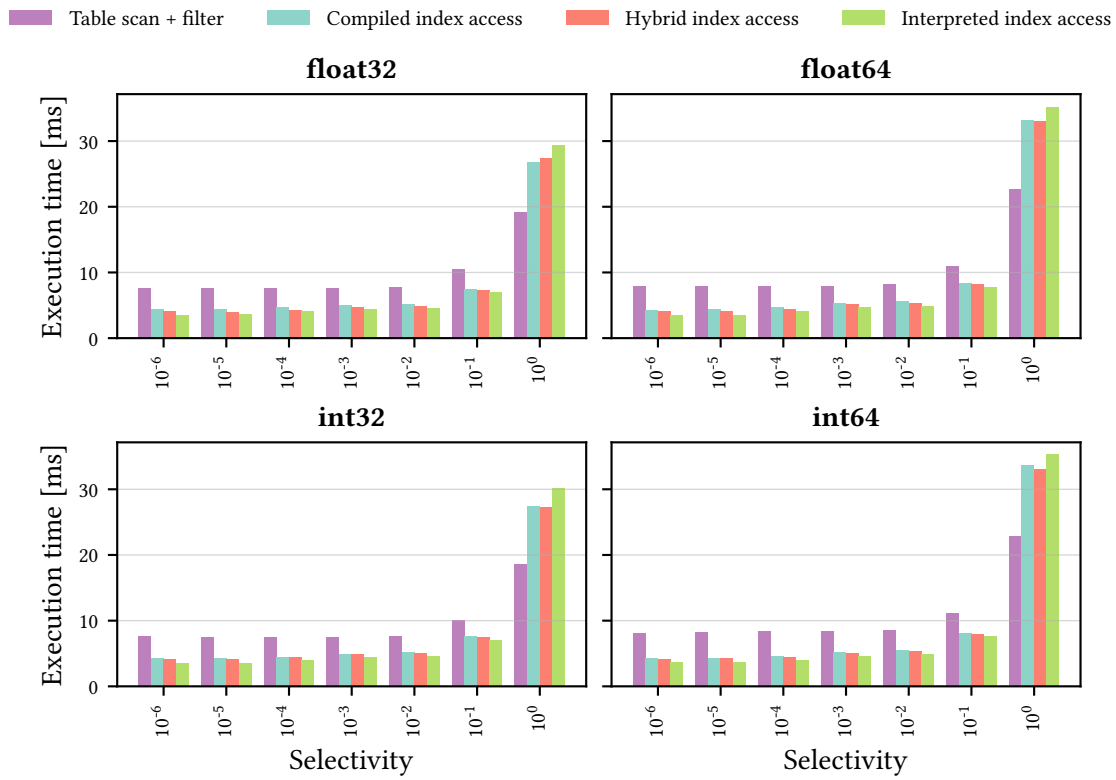


Figure 3.9: Execution time for the fastest variant of each strategy across different query selectivities.

strategies profit from their ability to retrieve qualifying tuple IDs in batches via calls to the host, resulting in better caching behavior. When we ran the experiment on different hardware, we generally observed the same trends, albeit with different absolute execution times and slightly different thresholds for the best-performing strategies due to different cache sizes.

3.6.3 Configuring the Compiled Index Access Strategy

In this experiment, we analyze the compiled index access strategy to determine the optimal batch size. Figure 3.10 compares execution times for varying batch sizes.

For selectivities of 10^{-3} (1k qualifying tuples) and higher, execution times remain nearly identical across different batch sizes. Although smaller batch sizes require more host calls for retrieving the qualifying tuple IDs, the overall execution time is primarily dominated by the compilation time to machine code at these high selectivities. Additionally, our implementation

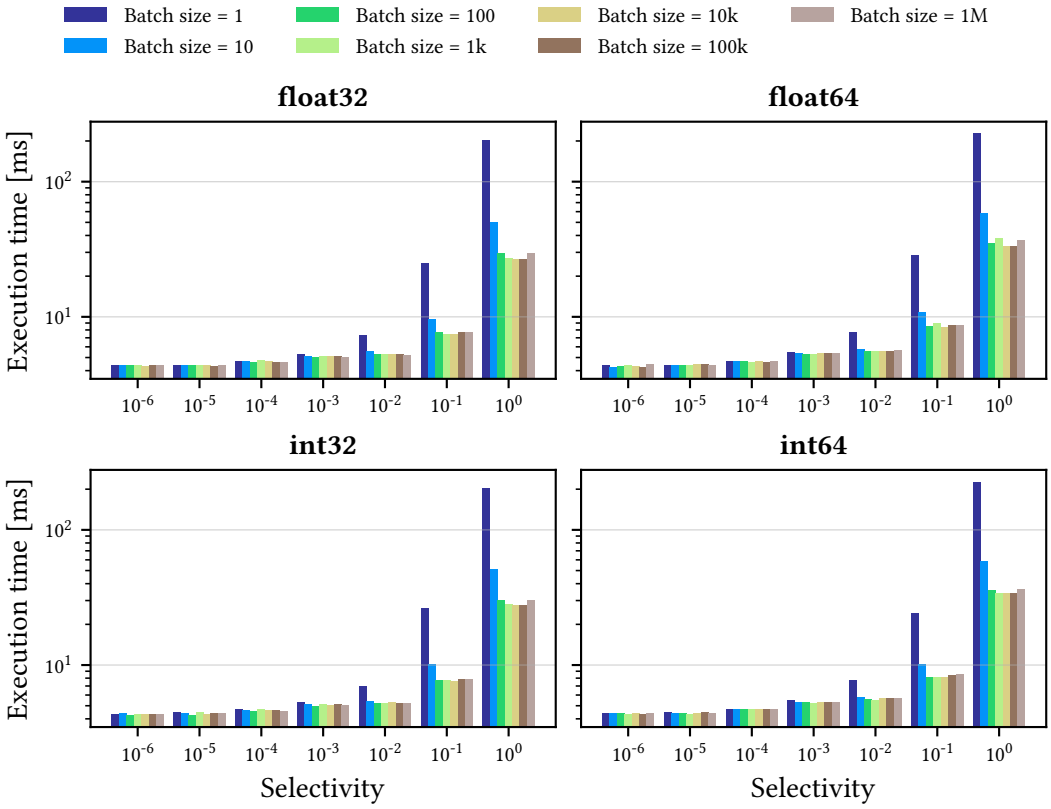


Figure 3.10: Execution time for the compiled index access strategy across different batch sizes.

allocates at most as much memory as is necessary for all qualifying tuple IDs, meaning that larger batch sizes effectively allocate the same amount of memory and make a single host call.

At selectivities of 10^{-2} (10k qualifying tuples) and lower, we observe a noticeable increase in execution time for smaller batch sizes, particularly for batch sizes of 1 and 10. Here, the compilation time to machine code no longer dominates the execution time, and the increased number of host calls becomes a substantial factor.

Contrary to our initial expectation, the largest batch size, which retrieves all tuple IDs in a single host call, does not perform the best. Instead, batch sizes of 1k to 100k, which require multiple host calls, exhibit superior performance. We hypothesize that these batch sizes achieve better caching behavior when transferring the tuple IDs from the host to the embedded runtime, as a batch of tuple IDs fits well into L1 cache. This observation demonstrates that there is a trade-off between the number of host calls and efficient cache usage. In summary,

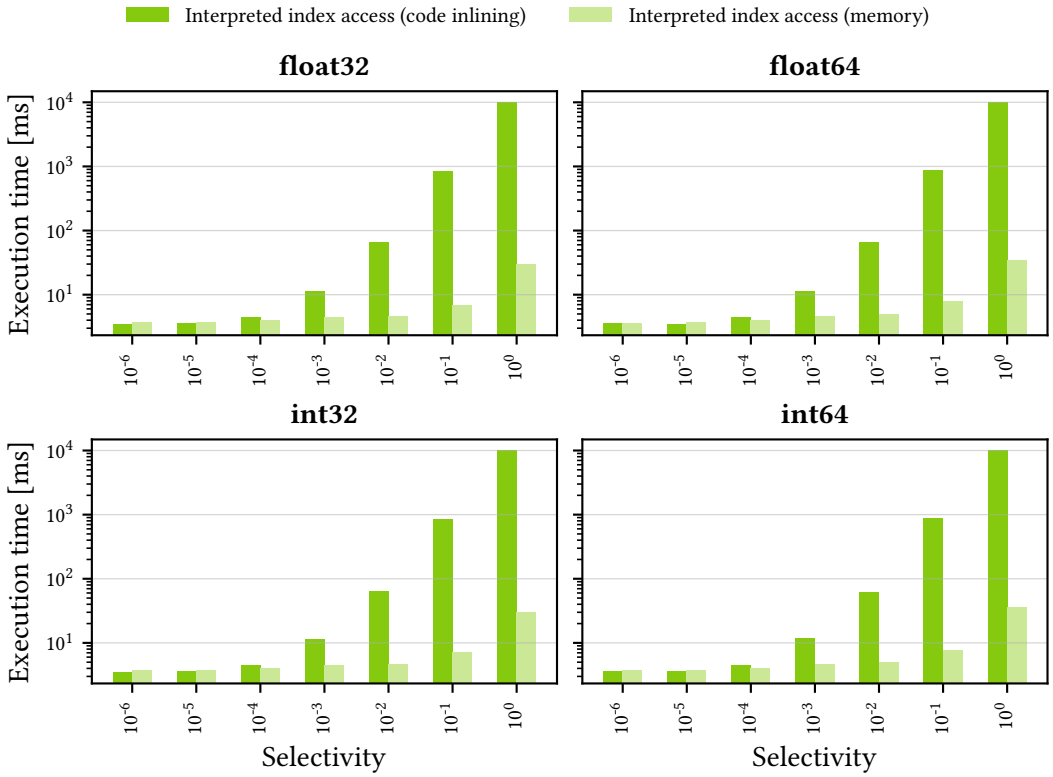


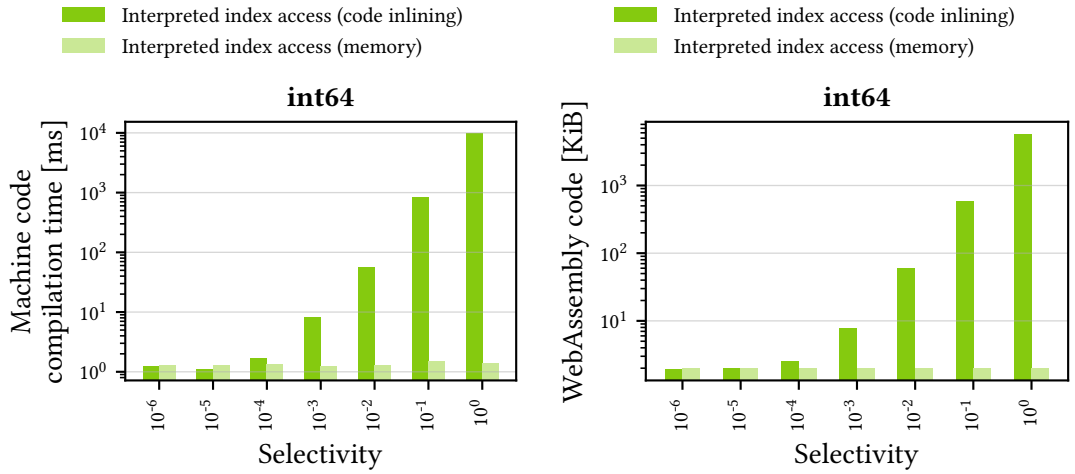
Figure 3.11: Execution time for the interpreted index access strategy across different materialization techniques.

batch sizes between 1k and 100k performed best, though these results may vary with different hardware.

3.6.4 Configuring the Interpreted Index Access Strategy

When using the interpreted index access strategy, we have two options for materializing qualifying tuple IDs: inlining them in the code or writing them to the WebAssembly memory and reading them at runtime. This experiment aims to determine which technique is preferable. Figure 3.11 compares execution times for both materialization techniques.

Both techniques exhibit similar performance for high selectivities, with inlining being slightly faster. This slight advantage is due to faster compilation times to machine code, which are particularly notable at higher selectivities, where compilation constitutes the majority of execution time. The faster compilation is caused by reduced code complexity, as our



(a) Machine code compilation time

(b) WebAssembly code size

Figure 3.12: Compilation time to machine code and WebAssembly code size for the interpreted index access strategy on the 64-bit integer dataset across different materialization techniques.

implementation of inlining eliminates the loop over the qualifying tuple IDs, as described in Section 3.5.2.

However, this changes significantly as the selectivity decreases. Starting at a selectivity of 10^{-4} (100 qualifying tuples), the execution time of inlining escalates dramatically. The main reason for this slower execution time is a sharp increase in compilation time compared to materialization in memory, as shown in Figure 3.12a for the 64-bit integer dataset. Figure 3.12b illustrates the corresponding WebAssembly code sizes for both materialization techniques.

For inlining, the size of the WebAssembly code grows linearly with the number of qualifying tuples, because with decreasing selectivity, more tuple IDs are inlined in the code. This larger code volume requires more extensive optimization by V8, leading to longer compilation times. Experimentation with larger datasets even caused V8 to crash due to exceeding the maximum permitted function size.

In contrast, when materializing the tuple IDs in memory, the code size is independent of the number of qualifying tuples and only differs in the number of loop passes required to read the tuple IDs from memory. Consequently, the compilation time remains near constant across different selectivities. In summary, when using the interpreted index access strategy, qualifying tuple IDs should be materialized in memory. Inlining is only viable when very few tuples qualify, typically in the tens.

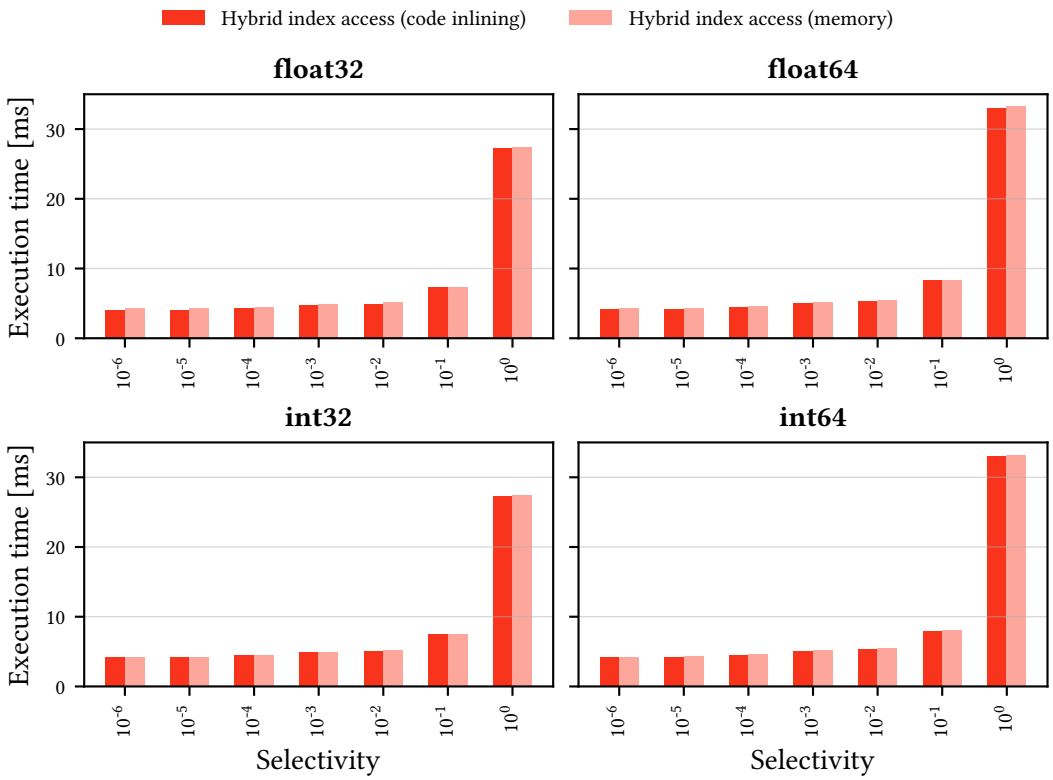


Figure 3.13: Execution time for the hybrid index access strategy across different materialization techniques using the fastest batch size.

3.6.5 Configuring the Hybrid Index Access Strategy

When using the hybrid index access strategy, we need to select both a materialization technique for the index offsets and an appropriate batch size. This experiment aims to identify the optimal configuration for both parameters. Let us first consider the materialization technique. Figure 3.13 shows the execution times for both materialization techniques using the best-performing batch size. Execution times are almost identical, but inlining generally performs slightly better. Unlike the interpreted index access strategy, the hybrid index access strategy only inlines two index offsets, making the code size independent of the number of qualifying tuples. Furthermore, when inlining, the number of loop passes is known at compile time, allowing V8 to better optimize the code, which likely causes the slightly faster execution times.

Given that inlining should generally be preferred when using the hybrid index access strategy, we now turn to the batch sizes. Figure 3.14 depicts execution times using inlining for varying

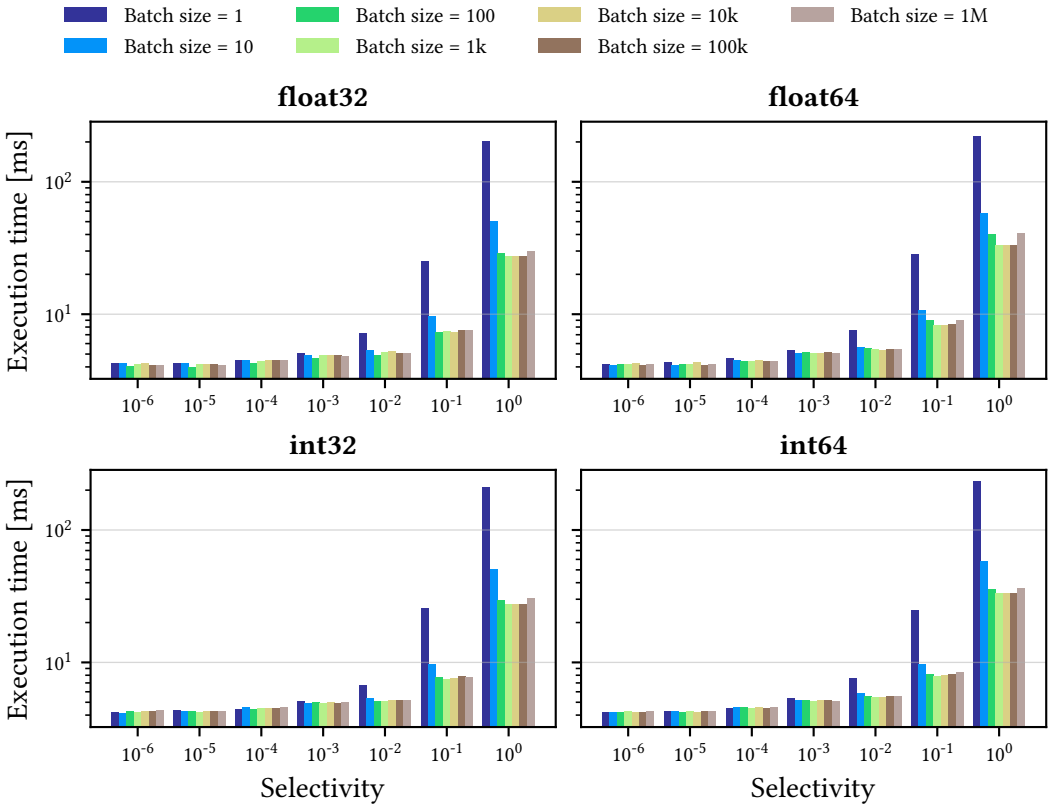


Figure 3.14: Hybrid index access strategy execution times with varying batch sizes using inlining.

batch sizes. As with the compiled index access strategy, we observe a trade-off between the number of host calls and efficient cache utilization. Again, batch sizes between 1k and 100k performed best, which is in line with the L1 cache size. In summary, the hybrid index access strategy performs best with inlined index offsets and a batch size that makes efficient use of the available cache.

3.6.6 Choosing an Index

Up to this point, our experiments have exclusively utilized the sorted array as the index. In the next experiment, we aim to explore the potential advantages of employing a more advanced index, such as an RMI. For this purpose, we compare the execution times of all strategies using both the sorted array and the RMI. Each strategy is optimally configured: the interpreted index access strategy materializes the tuple IDs in memory, the compiled index access strategy uses

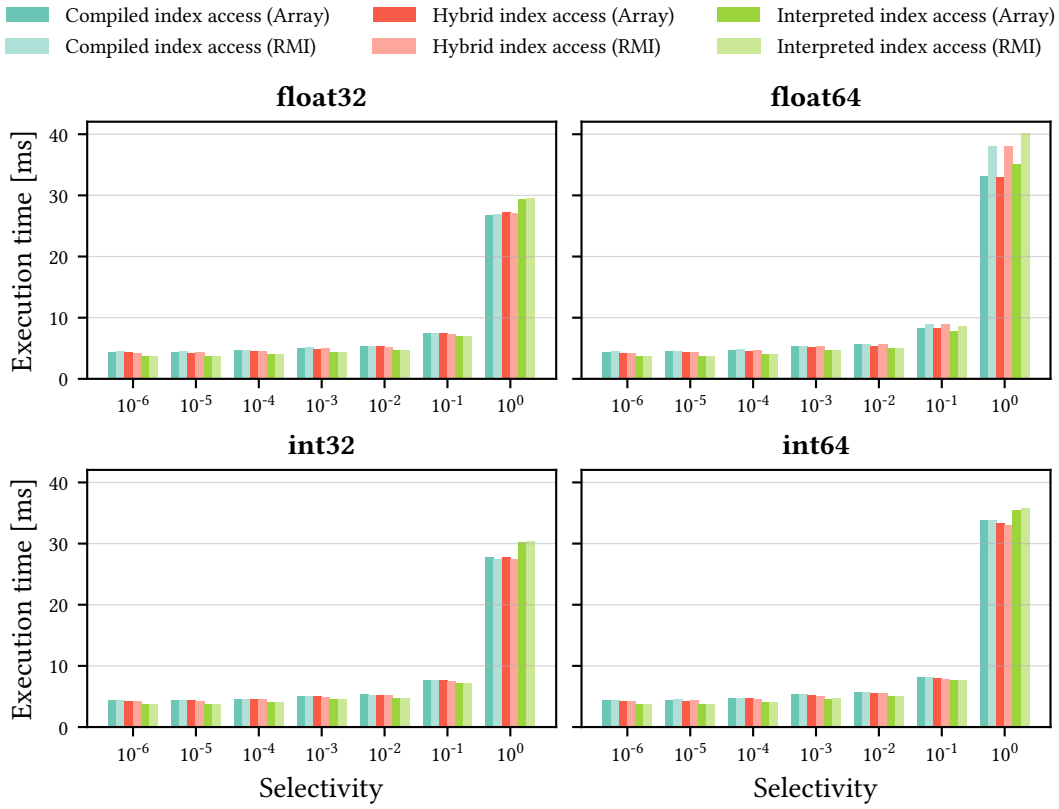


Figure 3.15: Execution times of the interpreted, hybrid, and compiled index access strategies when using either a sorted array or an RMI for the index scan.

a batch size of 10k, and the hybrid index access strategy materializes tuples using inlining with a batch size of 10k. The results are depicted in Figure 3.15.

Overall, the sorted array and the RMI exhibit almost identical performance across strategies and selectivities, with one exception: the 64-bit floating point dataset at selectivities 10^{-1} and 10^{0} (100k and 1M qualifying tuples, respectively). Here, the execution times using an RMI are several milliseconds slower. Unfortunately, we cannot explain this difference in execution times, and further research is required to understand the underlying cause.

The similar performance on the other datasets and selectivities is explained by the fact that only a tiny fraction of the execution time in the hundreds of nanoseconds is actually spent traversing the index to determine the offsets. The majority of the time is spent scanning the entries, loading the corresponding tuples from the store, and materializing the tuples as query result. While the index traversal time increases if the index has more entries, this time is

still negligible compared to the rest of the query execution. Consequently, advanced indexes should primarily be used to implement operators that require frequent index traversal, such as join operators or checking primary key constraints. In the context of index scans, it is more important that contiguous index entries can be scanned efficiently.

3.6.7 Benefiting From Caching Compiled Plans

In the previous experiments, we explicitly disabled caching of compiled WebAssembly modules in V8 to obtain stable results. The next experiment examines whether the strategies benefit from caching compiled plans when processing similar queries. For each dataset, we generate five queries (Q0 to Q4) with arbitrary selectivity, following the same format as the queries in previous experiments. We execute these queries with caching of compiled modules enabled. The compiled and interpreted index access strategies were configured optimally: the compiled index access strategy uses a batch size of 10k, the interpreted index access strategy materializes tuple IDs in memory. For the hybrid index access strategy, we consider both materialization techniques with a batch size of 10k. The rationale for considering both materialization techniques will be elucidated when considering the results. For this experiment, we focus on the compilation time from WebAssembly to machine code, as our goal is to improve this compilation time through reusing cached modules.

Figure 3.16 presents the results of this experiment. The baseline and all strategies, except the hybrid index access strategy with materialization in memory, exhibit similar compilation times for all five queries. This consistency indicates that these strategies do not benefit from caching.

The compiled index access strategy exhibits the slowest compilation times. It generates code with a nested loop: the outer loop performs the host calls to retrieve new tuple IDs, and the inner loop iterates over the tuple IDs to load the corresponding tuples from the store. In contrast, the baseline and the interpreted index access strategy produce simpler code with a single loop, iterating over all tuple IDs or qualifying tuple IDs, respectively. V8 requires less optimization time for this simpler code.

Similar to the compiled index access strategy, the hybrid index access strategy also generates WebAssembly code with a nested loop, leading to comparably slow compilation times for the first query Q0. When using inlining, the compilation time of subsequent queries remains high. However, when materializing the offsets in memory, subsequent queries exhibit significantly faster compilation times. This suggests that V8 caches the compiled module for the first query and reuses it for subsequent queries. Since the offsets were stored in memory, the generated

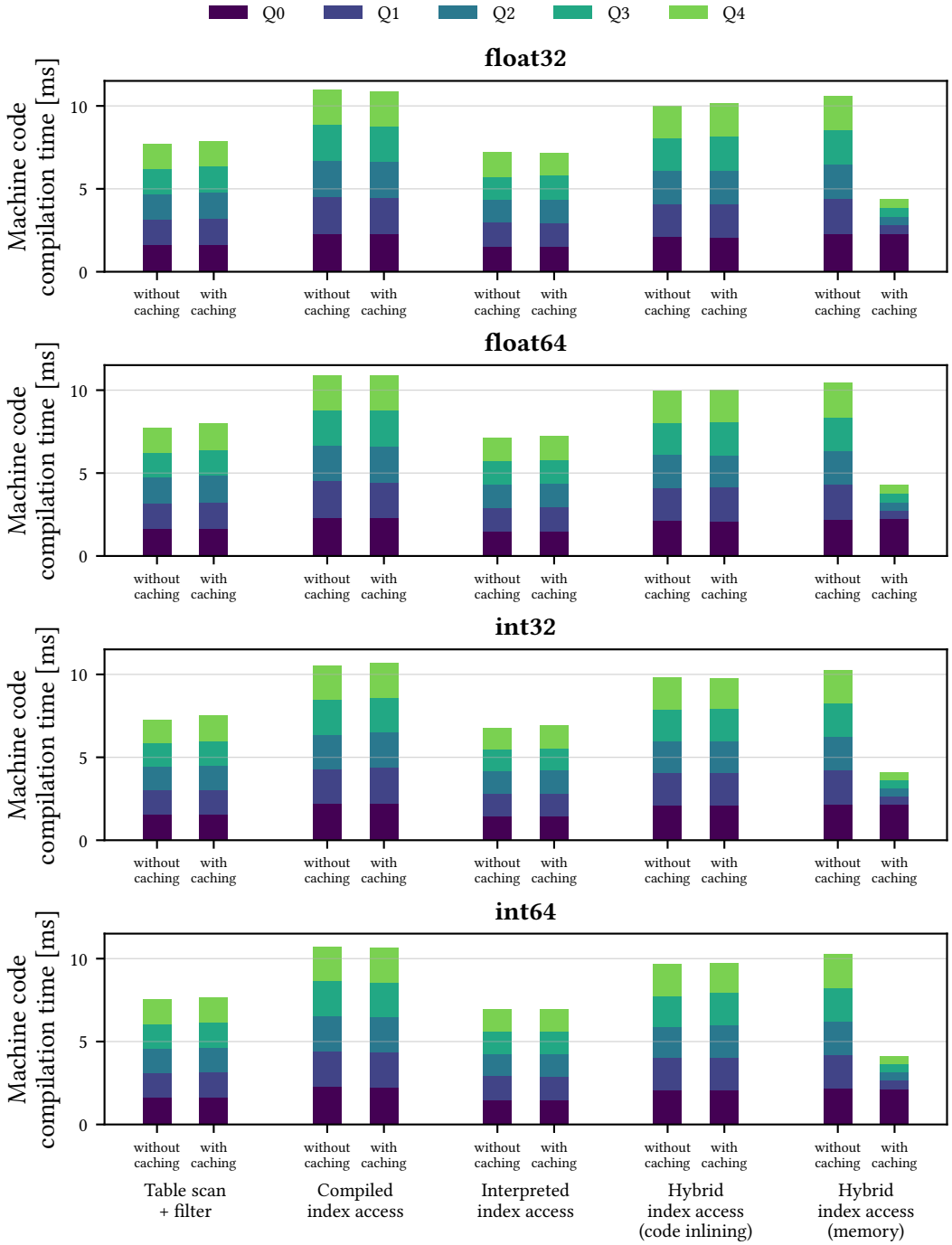


Figure 3.16: Machine code compilation times for five consecutive similar queries across five index access strategies, with and without caching of compiled WebAssembly modules in V8.

WebAssembly code is independent of the filter predicate of the query, resulting in identical generated code for queries with similar filter conditions.

Although the hybrid index access strategy with materialization in memory does not achieve the fastest execution times overall, it demonstrates the unique advantage of benefiting from caching compiled modules without additional effort. This feature may prove advantageous in scenarios where similar queries, differing only in filter conditions, are frequently executed.

3.7 Related Work

In the following, we review related work on index scan execution in compiling database systems. We emphasize that compiling database engines commonly utilize integrated execution using the compiled index access strategy and highlight various techniques to achieve this integrated execution (Section 3.7.1). Furthermore, we compare the index access strategies to adaptive query execution (Section 3.7.2). Lastly, we discuss related approaches for reusing compiled query plans (Section 3.7.3).

3.7.1 Index Scans in Compiling Database Systems

Native Execution Systems – With the rise in popularity of compiling approaches to query execution, a variety of such systems have been developed. Virtually all of these systems compile queries into executable code that runs within the same environment as the database itself. However, these systems take different paths to achieve this integrated execution. While the term *compiled index access strategy* is specific to our work, it effectively captures the essence of these systems' approach to accessing indexes as part of an index scan. Below, we explore three notable systems that exemplify this strategy in greater detail.

HIQUE [38] is one of the earliest compiling database systems that translates QEPs into C source files by assembling and specializing prewritten operator templates. The generated source file is then compiled into a shared library file using an external compiler and dynamically linked at system runtime. The query is executed by invoking a function in the dynamically loaded shared library, achieving execution in the same environment and enabling direct index access.

One notable drawback of HIQUE is the slow compilation times associated with optimizing C/C++ compilers. To mitigate these lengthy compilation times, systems have increasingly chosen to directly translate queries into a suitable intermediate representation. HyPer [51] relies on the Low Level Virtual Machine (LLVM), which is an open-source compiler infrastructure project designed for optimization, compilation, and execution of programs written in

various programming languages. HyPer translates QEPs into LLVM's intermediate representation (LLVM IR) using its own dedicated compiler. Complex algorithms and data structures, such as indexes, are written in C++ and compiled alongside the system. Conveniently, C++ methods can be invoked directly from LLVM IR, facilitating seamless access to indexes within the generated code. Essentially, the generated LLVM IR defines the control and data flow by connecting precompiled C++ code. Subsequently, a carefully selected subset of LLVM optimization passes is applied to the IR before it is compiled to executable machine code through LLVM's machine code generator and executed in the same environment.

Umbra [30] takes a further step by using its own intermediate representation, Umbra IR, specifically designed to optimize reads and writes, albeit with reduced flexibility for IR transformations compared to LLVM IR. Unlike LLVM IR, Umbra IR does not allow for directly calling C++ functions. Instead, Umbra employs a system of proxies that acts as an interface between Umbra IR and C++. The proxies eliminate the necessity of writing code to generate Umbra IR for each functionality. Developers can alternatively implement functionality in C++ and call it from the generated Umbra IR code via the proxies. Although not explicitly stated by the authors, it is likely that proxies are employed for accessing indexes and thus for performing index scans.

In conclusion, HIQUE, HyPer, and Umbra employ different techniques to achieve integrated execution, enabling direct access to tables and data structures like indexes. Each of these systems generates code that directly accesses indexes at query runtime, aligning with our definition of the compiled index access strategy.

JVM-Based Systems – Some database systems run compiled queries within the Java Virtual Machine (JVM). While this sounds like execution in an isolated environment at first, these systems are typically written in Java or Scala themselves and employ techniques to achieve execution within the same JVM environment as the system, allowing for a shared view on the data. Below, we briefly discuss three notable systems.

JAMDB [55] is a Java-based main-memory database prototype that translates QEPs into Java classes. These Java classes are then dynamically loaded and executed via Java reflection in the host environment, achieving integrated execution.

Apache Spark [3] is primarily a data analytics engine, not a database management system, and as such does not support indexes. However, Apache Spark's Tungsten engine [1] translates queries into Java code and uses Janino [62], a light-weight Java compiler, to compile this

generated code into Java classes. These classes are then dynamically loaded through Janino's custom class loader to be executed within the same JVM as Apache Spark.

Babelfish [24] is a query engine for polyglot queries, that again defines its own intermediate representation, Babelfish-IR. After optimizing the query, Babelfish leverages Truffle [65] and Graal Compiler [22] to compile the query into machine code and install said machine code directly into the JVM.

In conclusion, the discussed systems execute queries in the JVM but achieve integrated execution by dynamically loading compiled code into the host environment. This approach clearly distinguishes them from mu*t*able, where isolated execution is inevitable.

3.7.2 Adaptive Query Execution

Given that both the hybrid and interpreted index access strategies transition from interpreted to compiled execution, there is a natural proximity to adaptive query execution. We briefly introduce adaptive query execution and contrast it with these two strategies below.

Query compilation incurs a significant upfront cost, adding notable latency, especially for short-running queries. To alleviate this, Kohn, Leis, and Neumann [35] propose that a compiling query engine should support three execution modes: interpretation for short-running queries, optimizing compilation for long-running queries, and unoptimized compilation as a trade-off between the two.

Their architecture, implemented in HyPer, employs a model called morsel-wise execution [40] to adaptively switch between these modes. Initially, the QEP is translated into LLVM IR, and an LLVM bytecode generator produces the corresponding bytecode, which is then interpreted. During this interpreted execution, the system monitors the execution progress to determine if switching to compiled execution would be beneficial. If compiled execution is deemed advantageous, the system starts the compilation process in the background. Once compilation is complete, execution seamlessly transitions to the compiled code.

In follow-up work, Kersten, Leis, and Neumann further refine their approach, which is implemented in Umbra [30]. Recognizing that interpreted execution remains excessively slow, they decide to eliminate it. While keeping the optimizing compilation through LLVM, they replace the unoptimized compilation through LLVM with a fast just-in-time (JIT) compilation method using their own Umbra IR and a custom compiler named Flying Start. The objective of this JIT compilation is to quickly generated slightly optimized machine code.

Both adaptive execution and the index access strategies involve transitioning between execution modes, either from interpreted to compiled execution (HyPer and index access strategies) or from unoptimized to optimized compiled execution (Umbra). However, there are several key differences between the two.

First, as implied by the name, adaptive query execution dynamically shifts from one execution mode to another depending on runtime observations. In contrast, the index access strategies follow a static transition approach, switching either after determining the index offsets (hybrid index access strategy) or obtaining the qualifying tuple IDs (interpreted index access strategy).

Second, when adaptive query execution transitions to another execution mode, both are based on the same generated code. The index access strategies use interpretation during query compilation and use compiled execution during the entire runtime of the compiled query. This approach allows for integrating information, such as the exact number of qualifying tuples, into the generated code, thereby enabling compiler optimizations.

Third, the morsel-wise execution allows adaptive query execution to switch execution modes between the execution of two pipelines. In contrast, the index access strategies switch execution modes after executing a QEP fragment, or more specifically, while executing the index scan operator.

In the case of an isolated execution environment, unlike the interpreted index access strategy, which leveraged host-side index access during query compilation to avoid host calls, adaptive execution cannot eliminate these calls. This is because, with adaptive query execution, the compiled query would be executed entirely within the isolated environment of the embedded runtime, regardless of the execution mode.

3.7.3 Reusing Cached Plans

In Section 3.6.7, we observed that the hybrid index access strategy with offset materialization in memory simplifies caching compiled queries by rendering the generated code independent of constants in the filter expression. As a result, queries differing only in the filter predicate of the index scan can reuse previously compiled QEPs and avoid recompilation. While we consider this an incidental benefit of the strategy, there are systems that employ techniques to achieve this effect explicitly.

SingleStore [58], formerly known as MemSQL, transforms incoming queries into so-called parameterized queries by stripping out numeric and string constants [57]. SingleStore caches compiled parameterized queries so that subsequent queries sharing the same parameterized

structure reuse the cached version instead of triggering a new compilation process. Similarly, MapD [60], a GPU-accelerated database system, uses a comparable technique to extract constants and avoid recompilation.

These techniques are also closely related to prepared statements – parameterized SQL queries that are optimized and compiled once. Subsequently, these queries can be executed with different parameters without needing recompilation. The key difference is that in a prepared statement, the query must be explicitly parameterized, whereas the approaches above automatically parameterize the query. The hybrid index access strategy goes a step further, as the generated code inherently becomes independent of the constants in the filter condition.

3.8 Conclusion and Future Work

We introduced a novel query engine architecture that allows for partial execution of QEPs during the compilation process. Based on this architecture, we developed three distinct strategies for accessing indexes as part of an index scan within a compiling database system. Each strategy is characterized by when and how the index is accessed during query execution. While the compiled index access strategy interacts with the index solely at query runtime, the interpreted index access strategy retrieves tuple IDs of qualifying tuples during query compilation, incorporating them into the generated code. The hybrid index access strategy combines elements of both, conducting lookups on the index to obtain index entry offsets during query compilation but deferring the actual retrieval of tuple IDs until query runtime.

Our experimental evaluation indicates that none of the strategies universally outperforms the others. Instead, the optimal strategy depends on specific conditions. The interpreted index access strategy demonstrates efficacy when dealing with very few qualifying tuples, while the compiled index access strategy excels under lower selectivities, leveraging its ability to retrieve tuple IDs in batches to enhance caching efficiency. The hybrid index access strategy offers the advantage of generating code independent of the filter predicate on the indexed column, facilitating easy caching and reuse of compiled plans, thereby avoiding recompilation for queries differing only in these filter predicates.

In future work, we plan to broaden our evaluation scope by implementing strategy variants that access the index through exposed host memory, thereby directly interacting with the index without intermediary host calls. Our approach of executing parts of a query by interpretation during query compilation offers unique opportunities for augmenting the generated code. Thus, we propose exploring the feasibility of extending our approach to other operators.

Chapter 4

Conclusion

4.1 Summary

The first project, detailed in Chapter 2, provides an in-depth investigation of RMIs and its various hyperparameter's impact on prediction accuracy, lookup time, and build time. Our research led to the development of a practical guideline for configuring RMIs, achieving near-optimal performance on datasets without outliers. Outliers pose significant challenges for RMIs by increasing the range of values and affecting the tail behavior of the CDF. Simple models like linear regression, which treat each data point equally, often result in ineffective partitioning, with most data points falling into a single partition. While the reference implementation shows that good performance is still achievable, automating this process requires advanced data analysis to select an appropriate model before building the RMI in order to avoid exhaustive enumeration.

Our comparison of indexes supports the claims made in the original paper. RMIs indeed outperform traditional indexes such as B-trees and ART in terms of pure lookup performance, while also being significantly more space-efficient. However, this performance gain comes with trade-offs: RMIs have longer build times and limited applicability due to their immutability, making them primarily suitable for read-only scenarios. It is worth noting that some progress has been made on learned indexes that also support efficient updates (e.g. [67, 20, 41]), thereby expanding applicability of learned indexes to more diverse workloads. Other first-generation learned indexes, like RadixSpline or PGM-index, limit prediction error to optimize error correction time. In contrast, RMIs prioritize fast predictions to optimize evaluation time at the expense of potentially slower error correction. While this approach is effective when

prediction accuracy is high, it makes the lookup performance of RMIs less robust compared to other learned indexes.

The second project, detailed in Chapter 3, introduces a novel compiling query engine architecture that allows for partially executing query plans during compilation to integrate runtime observations into the generated code. Based on this architecture, we introduce three strategies for accessing indexes in an index scan. The compiled index access strategy generates code to access the index at query runtime, similar to how most systems operate. The interpreted index access strategy performs index accesses during query compilation, integrating the qualifying tuple IDs into the generated code, thus eliminating any need to interact with the index at query runtime. The hybrid index access strategy combines elements of both previous strategies, looking up offsets into the index during query compilation and scanning the entries at query runtime.

Our experimental evaluation reveals that no single strategy consistently outperforms the others; the best-performing strategy varies depending on the number of qualifying tuples. The interpreted index access strategy proves most effective for highly selective queries. These queries execute quickly, and the reduced compilation time, due to the simpler code structure, is particularly notable. In contrast, the compiled index access strategy excels with lower selectivities, where compilation time becomes negligible compared to overall query execution time, benefiting from positive caching effects through batching. The hybrid index access strategy is particularly effective in scenarios where compiled plans are cached and queries share a similar format. By generating code that only refers to offsets in the index rather than specific filter predicates, the hybrid index access strategy can reuse previously compiled plans for similar queries, eliminating the need for compilation.

We also analyzed the strategies with multiple indexes, including RMIs, to determine the extent to which RMIs affect the performance of index scans. Although exhibiting exceptional performance in isolation, RMIs could not improve the performance of index scans compared to a simple baseline index. The main reason for this observation is that the index access time (hundreds of nanoseconds) accounts for roughly 1/100th of the overall query processing time (tens of microseconds), making its impact on performance negligible in the context of the total query execution time.

4.2 Discussion

In our research, we investigated RMIs from both a component view and a system view, each contributing to a comprehensive understanding of RMIs under different conditions. The component view allowed us to evaluate RMIs in isolation, providing insights into their maximum potential under ideal conditions, free from system complexities and external factors. Conversely, the system view assessed RMIs within a real database system as part of an index scan operator, offering a glimpse into their practical performance in real-world applications.

Our experiments uncovered a notable discrepancy between RMIs' performance in isolated conditions versus in real systems. While RMIs demonstrate superior performance over baselines like binary search or B-trees in isolated scenarios, they did not outperform binary search over a sorted array in index scan operations. This disparity primarily stems from two reasons: first, the lookup time of an index is negligible compared to the overall query processing time, resulting in only marginal potential improvement; second, the index scan operator performs only two lookups, which is insufficient for the CPU to make optimizations like partially caching the RMI.

Moreover, our findings caution against blindly replacing traditional indexes with RMIs in real systems expecting automatic performance gains. Instead, the decision to integrate RMIs should consider the specific workload and application context. Due to their lack of support for updates, RMIs are unsuitable for OLTP workloads but may benefit OLAP workloads, particularly in read-only scenarios where fast lookup times are advantageous.

Furthermore, our research highlights that RMIs may not fully leverage their enhanced space efficiency and lookup performance in scenarios with few consecutive lookups, such as index scans. However, we anticipate better performance in contexts like index nested loop joins, where RMIs are queried frequently in short intervals. These insights underscore the importance of context-specific evaluations and emphasize the need for further research to explore the full potential and limitations of learned indexes across diverse database applications.

4.3 Limitations

While our research has provided valuable insights into the performance and applicability of RMIs in database systems, it is important to recognize several limitations that constrain the generalizability and scope of our findings. These limitations highlight areas where future research can further expand and refine our understanding.

A Critical Analysis of Recursive Model Indexes

Scope of Hyperparamter Configurations – Despite investigating a total of 1280 hyperparameter configurations, our evaluation of RMIs was not exhaustive. We examined only four distinct model types: linear regression, linear spline, cubic spline, and radix. Notable exclusions are histograms, polynomial models, and linear models optimizing the logarithmic error [14]. However, we do not anticipate that these additional model types would significantly alter our primary conclusions, as these models also tend to struggle with outliers.

Focus on Early Learned Indexes – Our research primarily focused on RMIs, the first learned range index. We compared RMIs to other first-generation learned indexes and traditional indexes. Consequently, our conclusions might not extend to more recent learned indexes that incorporate additional features, such as support for efficient updates and concurrent queries. While studies including newer learned indexes exist, these studies use the reference implementation of RMIs rather than our improved implementation [19, 61].

Index Access Strategies for Index Scans

Emphasis on Index Scan – Our study integrated RMIs into the mu*t*able database system and utilizes them in the index scan operator. While results indicate, that RMIs are not effective for that particular use case in a real system, this does not imply that RMIs are ineffective in general.

Partial Strategy Implementation – Due to time constraints, our evaluation of the index access strategies did not include implementations that perform the index access within the embedded runtime by making the memory region containing the index available to the WebAssembly module through its linear memory. This implementation of the hybrid and compiled index access strategies would eliminate host calls at the cost of setting up the linear memory with the index prior to execution. Implementing these variants is crucial to exploring this trade-off and providing a complete picture.

Evaluation Environment – We evaluated the strategies in a highly controlled environment, where execution was single-threaded using synthetic datasets and workloads that aimed to isolate the performance impact of the index scan as much as possible. While we expect our primary observations to still hold in a more complex system under diverse workloads, our experiments do not allow us to confirm this with certainty.

Specificity of Database System – The strategies were developed with the execution model of mu*t*able in mind and subsequently implemented in mu*t*able. Consequently, our results are

specific to mu*t*able. Although these strategies can be transferred to and implemented in other database systems, those systems might produce different performance results, due to differing architectures and optimizations.

By addressing these limitations, future research can build on our findings and further elucidate the conditions under which RMIs can be most effectively utilized. This will help bridge the gap between theoretical potential and practical application, ensuring that learned indexes can be effectively integrated into a variety of database environments.

4.4 Future Research Directions

Given the insights and limitations outlined above, several avenues for future research emerge. These directions aim to deepen our understanding of learned indexes, address challenges identified in our current work, and explore the application of concepts in other contexts.

Addressing Outliers by Extending Hyperparameter Analysis – Extreme outliers significantly impact the tail behavior of the CDF, posing challenges for RMIs and leading to suboptimal lookup times across the evaluated configurations. This issue primarily stems from the partitioning behavior of the model types used in the first layer of the RMIs. While these models are efficient in training and evaluation, they often fail to accurately capture the complex shapes of the CDF, resulting in skewed data distribution among the models in the second layer.

To mitigate this, future research should focus on expanding the hyperparameter analysis to include a broader range of model types. This expansion should encompass not only simpler models such as histograms and polynomial regression, but also more sophisticated approaches like shallow neural networks. More complex models are likely to be slower to evaluate but could offer better prediction accuracy, presenting a trade-off between evaluation time and error correction time. By exploring these diverse model types, we aim to achieve more robust partitioning of data points, thereby improving overall performance and adaptability of RMIs. Additionally, our configuration guidelines should be extended accordingly, empowering practitioners to effectively tailor RMIs to diverse and challenging datasets.

Expanding Index Scan Strategy Evaluation – We developed three strategies for accessing indexes in an index scan that can be applied to various execution models. For implementation and evaluation, we chose mu*t*able, which generates WebAssembly code and executes it in an embedded runtime. In this environment, the host memory containing the index cannot be accessed directly. Accessing host memory can be achieved through host calls or by explicitly exposing host memory to the WebAssembly program running in the embedded runtime.

Due to time constraints and high engineering effort, we did not implement access through exposed host memory. This approach involves not only creating a framework for exposing the memory containing the indexes but also replicating the index access logic in WebAssembly, as the program operates directly on the raw memory containing the index. Despite these challenges, implementing and evaluating this method is essential for a comprehensive understanding of the index strategies. This variant is particularly important as it closely resembles execution in an integrated environment used by most database systems.

Generalizing Strategies Across Database Operators – The interpreted and hybrid index access strategies leverage efficiency gains by executing parts of the index scan operator during query compilation. This approach not only simplifies the generated code by eliminating the index access logic but also facilitates compiler optimizations, such as loop unrolling through inlining of intermediate results. Moreover, avoiding the reimplementation of index access logic significantly reduces the engineering effort. These advantages suggest an opportunity to formalize and extend the strategies to other database operators. For instance, optimizing hash table construction in simple hash joins or handling index-only subqueries during query compilation are promising applications.

Integrating Learned Indexes Into Other Operators – We integrated RMIs into the index scan operator of mutable as part of our evaluation of index scan strategies. In our experiments, the use of RMIs did not enhance performance compared to binary search over a sorted array. This is primarily because the time spent traversing the index is negligible compared to the overall query processing time. To fully assess the performance of learned indexes in a real system context, they must be integrated and evaluated within other operators.

A promising candidate is the index nested loop join, where each tuple in the probe relation triggers an index lookup. These repeated lookups create a favorable access pattern for learned indexes, enabling caching and leveraging their space efficiency. However, for equality predicates, hash tables, often built on primary keys by default, are hard to beat. Thus, examining joins with range predicates, known as range joins, is crucial since hash tables do not support range predicates and cannot be applied here.

In conclusion, this thesis provides valuable insights into RMIs and index access strategies. However, these findings are just one piece of the puzzle in advancing the field of learned indexes within database systems. Further exploration will be crucial to unlocking their full potential.

Bibliography

[1] Sameer Agarwal, Davies Liu, and Reynold Xin. *Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop*. 2016. URL: https://www.databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html.

[2] Dana Van Aken et al. "Automatic Database Management System Tuning Through Large-scale Machine Learning." In: *SIGMOD Conference*. ACM, 2017, pp. 1009–1024.

[3] Apache Foundation. *Apache Spark – Unified Engine for large-scale data analytics*. 2024. URL: https://spark.apache.org/.

[4] Peter Balis et al. *Don't Throw Out Your Algorithms Book Just Yet: Classical Data Structures That Can Outperform Learned Indexes*. 2018. URL: https://dawn.cs.stanford.edu/2018/01/11/index-baselines/.

[5] Rudolf Bayer and Edward M. McCreight. "Organization and Maintenance of Large Ordered Indexes." In: *SIGFIDET Workshop*. ACM, 1970, pp. 107–141.

[6] Timo Bingmann. *TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers*. 2018. URL: https://github.com/tlx/tlx.

[7] Peter A. Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution." In: *CIDR*. www.cidrdb.org, 2005, pp. 225–237.

[8] Edgar Frank Codd. "A Relational Model of Data for Large Shared Data Banks." In: *Commun. ACM* 13.6 (1970), pp. 377–387.

[9] cppreference.com. *std::lower_bound*. 2024. URL: https://en.cppreference.com/w/cpp/algorithm/lower_bound.

[10] Andrew Crotty. "Hist-Tree: Those Who Ignore It Are Doomed to Learn." In: *CIDR*. www.cidrdb.org, 2021.

[11] Jialin Ding. *ALEX: A library for building an in-memory, Adaptive Learned indEX*. 2020. URL: https://github.com/microsoft/ALEX.

[12] Jialin Ding et al. "ALEX: An Updatable Adaptive Learned Index." In: *SIGMOD Conference*. ACM, 2020, pp. 969–984.

[13] Jialin Ding et al. "Instance-Optimized Data Layouts for Cloud Analytics Workloads." In: *SIGMOD Conference*. ACM, 2021, pp. 418–431.

[14] Martin Eppert, Philipp Fent, and Thomas Neumann. "A Tailored Regression for Learned Indexes: Logarithmic Error Regression." In: *aiDM@SIGMOD*. ACM, 2021, pp. 9–15.

[15] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. "Why Are Learned Indexes So Effective?" In: *ICML*. Vol. 119. PMLR, 2020, pp. 3123–3132.

[16] Paolo Ferragina and Giorgio Vinciguerra. "The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds." In: *Proc. VLDB Endow.* 13.8 (2020), pp. 1162–1175.

[17] Yoshihiko Futamura. "Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler." In: *High. Order Symb. Comput.* 12.4 (1999), pp. 381–391.

[18] Alex Galakatos et al. "FITing-Tree: A Data-aware Index Structure." In: *SIGMOD Conference*. ACM, 2019, pp. 1189–1206.

[19] Jiake Ge et al. "Cutting Learned Index into Pieces: An In-depth Inquiry into Updatable Learned Indexes." In: *ICDE*. IEEE, 2023, pp. 315–327.

[20] Jiake Ge et al. "SALI: A Scalable Adaptive Learned Index Framework based on Probability Models." In: *Proc. ACM Manag. Data* 1.4 (2023), 258:1–258:25.

[21] Google. *V8 JavaScript Engine*. 2024. URL: https://v8.dev/.

[22] GraalVM Project. *Graal Compiler*. 2024. URL: https://www.graalvm.org/latest/reference-manual/java/compiler/.

[23] Goetz Graefe and David J. DeWitt. "The EXODUS Optimizer Generator." In: *SIGMOD Conference*. ACM, 1987, pp. 160–172.

[24] Philipp Marian Grulich, Steffen Zeuch, and Volker Markl. "Babelfish: Efficient Execution of Polyglot Queries." In: *Proc. VLDB Endow.* 15.2 (2021), pp. 196–210.

[25] Immanuel Haffner and Jens Dittrich. "A Simplified Architecture for Fast, Adaptive Compilation and Execution of SQL Queries." In: *EDBT*. OpenProceedings.org, 2023.

[26] Immanuel Haffner and Jens Dittrich. "mutable: A Modern DBMS for Research and Fast Prototyping." In: *CIDR*. www.cidrdb.org, 2023.

[27] Immanuel Haffner et al. *mutable: A Database System for Research and Fast Prototyping*. 2022. URL: https://github.com/mutable-org/mutable.

[28] Benjamin Hilprecht et al. "DeepDB: Learn from Data, not from Queries!" In: *Proc. VLDB Endow.* 13.7 (2020), pp. 992–1005.

[29] Allen Huang et al. *Learned Index Leaderboard*. 2021. URL: https://learnedsystems.github.io/SOSDLeaderboard.

[30] Timo Kersten, Viktor Leis, and Thomas Neumann. "Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra." In: *Proc. VLDB Endow.* 30.5 (2021), pp. 883–905.

[31] Changkyu Kim et al. "FAST: fast architecture sensitive tree search on modern CPUs and GPUs." In: *SIGMOD Conference*. ACM, 2010, pp. 339–350.

[32] Andreas Kipf and Alexander van Renen. *RadixSpline: A Single-Pass Learned Index*. 2020. URL: https://github.com/learnedsystems/RadixSpline.

[33] Andreas Kipf et al. "SOSD: A Benchmark for Learned Indexes." In: *CoRR* abs/1911.13014 (2019).

[34] Andreas Kipf et al. "RadixSpline: a single-pass learned index." In: *aiDM@SIGMOD*. ACM, 2020, 5:1–5:5.

[35] André Kohn, Viktor Leis, and Thomas Neumann. "Adaptive Execution of Compiled Queries." In: *ICDE*. IEEE, 2018, pp. 197–208.

[36] Tim Kraska et al. "The Case for Learned Index Structures." In: *CoRR* abs/1712.01208v1 (2017).

[37] Tim Kraska et al. "The Case for Learned Index Structures." In: *SIGMOD Conference*. ACM, 2018, pp. 489–504.

[38] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. "Generating code for holistic query evaluation." In: *ICDE*. IEEE, 2010, pp. 613–624.

[39] Viktor Leis, Alfons Kemper, and Thomas Neumann. "The adaptive radix tree: ARTful indexing for main-memory databases." In: *ICDE*. IEEE, 2013, pp. 38–49.

[40] Viktor Leis et al. "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age." In: *SIGMOD Conference*. ACM, 2014, pp. 743–754.

[41] Pengfei Li et al. "DILI: A Distribution-Driven Learned Index." In: *Proc. VLDB Endow.* 16.9 (2023), pp. 2212–2224.

[42] Qingzhi Ma and Peter Triantafillou. "DBEst: Revisiting Approximate Query Processing Engines with Machine Learning Models." In: *SIGMOD Conference*. ACM, 2019, pp. 1553–1570.

[43] Marcel Maltry. *Code for our VLDB paper: A Critical Analysis of Recursive Model Indexes*. 2021. URL: https://github.com/BigDataAnalyticsGroup/analysis-rmi.

[44] Marcel Maltry and Jens Dittrich. "A Critical Analysis of Recursive Model Indexes." In: *Proc. VLDB Endow.* 15.5 (2022), pp. 1079–1091.

[45] Ryan Marcus. *RMI: The recursive model index, a learned index structure*. 2019. URL: https://github.com/learnedsystems/RMI.

[46] Ryan Marcus, Andreas Kipf, and Alexander van Renen. *SOSD: A Benchmark for Learned Indexes*. 2019. URL: https://github.com/learnedsystems/SOSD.

[47] Ryan Marcus, Emily Zhang, and Tim Kraska. "CDFShop: Exploring and Optimizing Learned Index Structures." In: *SIGMOD*. ACM, 2020, pp. 2789–2792.

[48] Ryan Marcus et al. "Neo: A Learned Query Optimizer." In: *Proc. VLDB Endow.* 12.11 (2019), pp. 1705–1718.

[49] Ryan Marcus et al. "Benchmarking Learned Indexes." In: *Proc. VLDB Endow.* 14.1 (2020), pp. 1–13.

[50] Ryan Marcus et al. "Bao: Making Learned Query Optimization Practical." In: *SIGMOD Rec.* 51.1 (2022), pp. 6–13.

[51] Thomas Neumann. "Efficiently Compiling Efficient Query Plans for Modern Hardware." In: *Proc. VLDB Endow.* 4.9 (2011), pp. 539–550.

[52] Thomas Neumann. *The Case for B-Tree Index Structures*. 2017. URL: http://databasearchitects.blogspot.com/2017/12/the-case-for-b-tree-index-structures.html.

[53] Sriram Padmanabhan et al. "Block Oriented Processing of Relational Database Operations in Modern Computer Architectures." In: *ICDE*. IEEE, 2001, pp. 567–574.

[54] Andrew Pavlo et al. "Self-Driving Database Management Systems." In: *CIDR*. www.cidrdb.org, 2017.

[55] Jun Rao et al. "Compiled Query Execution Engine using JVM." In: *ICDE*. IEEE, 2006, p. 23.

[56] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. "RUMA has it: Rewired User-space Memory Access is Possible!" In: *Proc. VLDB Endow.* 9.10 (2016), pp. 768–779.

[57] SingleStore. *Documentation – Query Compilation*. 2024. URL: https://docs.singlestore.com/cloud/getting-started-with-singlestore-helios/about-singlestore-helios/singlestore-helios-faqs/query-compilation/.

[58] SingleStore. *SingleStore*. 2024. URL: https://www.singlestore.com/.

[59] Mihail Stoian and Andreas Kipf. *CHT: Implementation of the compact "Hist-Tree"*. 2021. URL: https://github.com/stoianmihail/CHT.

[60] Alex Suhan. *MapD: Massive Throughput Database Queries with LLVM on GPUs*. 2015. URL: https://developer.nvidia.com/blog/mapd-massive-throughput-database-queries-llvm-gpus/.

[61] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. "Learned Index: A Comprehensive Experimental Evaluation." In: *Proc. VLDB Endow.* 16.8 (2023), pp. 1992–2004.

[62] Arno Unkrig. *Janino – A super-small, super-fast Java compiler*. 2024. URL: http://janino-compiler.github.io/janino/.

[63] Giorgio Vinciguerra. *PGM-index: State-of-the-art learned data structure*. 2019. URL: https://github.com/gvinciguerra/PGM-index.

[64] Xiaoying Wang et al. "Are We Ready For Learned Cardinality Estimation?" In: *Proc. VLDB Endow.* 14.9 (2021), pp. 1640–1654.

[65] Christian Wimmer and Thomas Würthinger. "Truffle: a self-optimizing runtime system." In: *SPLASH*. ACM, 2012, pp. 13–14.

[66] Lucas Woltmann et al. "Cardinality estimation with local deep learning models." In: *aiDM@SIGMOD*. ACM, 2019, 5:1–5:8.

[67] Chaichon Wongkham et al. "Are Updatable Learned Indexes Ready?" In: *Proc. VLDB Endow.* 15.11 (2022), pp. 3004–3017.

[68] World Wide Web Consortium (W3C). *WebAssembly Core Specification (2nd Edition)*. 2024. URL: https://www.w3.org/TR/wasm-core-2/.

[69] Zongheng Yang et al. "Qd-tree: Learning Data Layouts for Big Data Analytics." In: *SIGMOD Conference*. ACM, 2020, pp. 193–208.

[70] Meifan Zhang and Hongzhi Wang. "LAQP: Learning-based approximate query processing." In: *Inf. Sci.* 546 (2021), pp. 1113–1134.