Improving Index Recommendation for MongoDB

YAN RONG

SID: 500514742

Supervisor: Prof. Alan Fekete
Associate Supervisor: Dr. Thomas Rueckstiess

This thesis is submitted in partial fulfillment of
the requirements for the degree of
Bachelor of Advanced Computing (Honours)

School of Computer Science
The University of Sydney
Australia

5 December 2024

THE UNIVERSITY OF
SYDNEY

Abstract

Indexing is essential in database systems, enabling efficient data retrieval and optimising query performance. However, index recommendation remains underexplored for NoSQL systems like MongoDB, unlike relational databases. Currently, there is no widely adopted or effective index recommender available for MongoDB. Existing tools, such as Mindexer, use simplified cost estimation models that overlook important factors, such as index structure, index size, and field ordering, leading to suboptimal recommendations. This study addresses these limitations by developing an enhanced index recommendation tool, Index Minder, which provides more accurate cost estimations and improved recommendations.

The primary aim of this research was to test the hypothesis that incorporating a comprehensive index usage cost model—accounting for scan cost, lookup effect, and sorting benefits—would result in better index recommendations and improved query performance.

The study followed an *engineering research methodology*, consisting of three phases: (1) performance analysis of Mindexer, (2) proposal of modifications, including the development of a novel Index Scan Cost Estimator (ISCE), and (3) performance evaluation of Index Minder through comparison with Mindexer using synthetic datasets and statistical tests such as t-tests and ANOVA.

The results demonstrate that Index Minder outperforms Mindexer by consistently delivering significant reductions in workload execution time, indicating that Index Minder recommends higher-quality indexes. Additionally, Index Minder maintains robust performance across different sample ratios, further validating the hypothesis that a comprehensive cost model enhances index recommendations.

This research makes a significant contribution to the field by advancing index recommendation strategies with a novel cost-based approach, addressing the limitations of existing tools. With further development, Index Minder holds the potential to become a versatile tool for mixed workloads and diverse database systems, improving query performance while adapting to various environments.

Acknowledgements

I would like to extend my heartfelt gratitude to those who have supported me throughout the journey of completing this thesis.

First and foremost, I am deeply grateful to my Supervisor, **Professor Alan Fekete**, for his invaluable guidance and encouragement. His mentorship extended beyond research supervision, enriching my academic life with a profound sense of the research culture. Through his support, I was able to engage fully with the research group, exploring both current research topics and future pathways. Professor Fekete's insight into academic career planning has been instrumental, and his advice was invaluable in my PhD application process. His unwavering support has inspired me every step of the way.

I would also like to thank my Associate Supervisor, **Dr. Thomas Rueckstiess**, for his generosity in providing technical guidance and access to Mindexer, an experimental index recommendation tool that has been crucial to this research. His deep technical knowledge and practical insights were essential in overcoming many challenges. Dr. Rueckstiess also provided valuable perspectives on both academic and industrial career paths, guiding me toward potential future directions.

My sincere thanks go to **Professor of Practice Michael Cahill** for his invaluable support in deepening my understanding of MongoDB's internal index mechanisms. His assistance and encouragement during my PhD application process were equally essential, and I am incredibly grateful for his insights and mentorship. I would also like to extend my gratitude to **Associate Professor Uwe Röhm** for the opportunity to contribute to database education through teaching, an experience that has profoundly enriched my academic journey and skills.

Finally, I wish to express my deepest appreciation to my parents for their unwavering support and unconditional love. They have encouraged me in every academic decision and provided both moral and financial support, making this journey possible. Their belief in me has been my greatest strength.

To everyone who contributed to this milestone in my life—thank you.

CONTENTS

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Background and Context

The proliferation of data in modern applications has necessitated the development of scalable and efficient data management solutions. **NoSQL databases**, such as **MongoDB** (MongoDB, Inc., 2024c), have become a popular choice for storing and processing massive volumes of unstructured data due to their flexibility, ease of scaling, and suitability for cloud environments. Among all NoSQL databases, **MongoDB** stands out as a leading option, ranking **fifth overall** in popularity across all database management systems, according to the **DB-Engines Ranking** (DB-Engines, 2024). Notably, it holds the **highest ranking score among all NoSQL databases** and is even close to **relational databases**, highlighting its increasing adoption in a variety of applications.

The high ranking of **MongoDB** implies a growing interest and development effort within the developer community, indicating that more and more developers are adopting MongoDB for data-intensive tasks. This trend emphasises the need for efficient **query optimisation** techniques, as the effective use of **indexes** directly affects the performance of MongoDB-based systems, making **index recommendation** a crucial area for performance improvement.

1.2 Research Problem and Motivation

Indexes are a crucial component in improving **query execution performance** by speeding up data retrieval, but not all indexes are beneficial. Poor index choices can lead to negative impacts on performance and incur significant **maintenance overhead** during **Create, Update, and Delete (CUD)** operations. This makes the task of recommending effective indexes complex and non-trivial, especially as databases grow in size and complexity.

Manual selection of indexes is **time-consuming** and requires significant expertise, making it impractical for large-scale, rapidly evolving systems. Therefore, a **reliable automated index recommender** is desired.

Existing tools, such as **Atlas Performance Advisor** and experimental systems like **Mindexer** (Rueckstiess, 2023), fall short in providing reliable recommendations. Moreover, while **relational database index recommendation** has been extensively studied, there is a significant **lack of research** on **index recommendation for NoSQL databases**, especially **MongoDB**. This gap makes it challenging to design effective index recommendation systems for NoSQL databases that can handle diverse and complex workloads.

The **research problem** addressed in this thesis is to **improve index selection for MongoDB** by developing a more accurate, reliable, and efficient index recommender compared to existing tools like **Atlas Performance Advisor** and **Mindexer**.

1.3 Main Contributions

The main contributions of this thesis are as follows:

(1) **Insights into the ESR Rule**: This research offers valuable insights into the rationale behind the **Equality-Sort-Range (ESR) rule**, recommended by **MongoDB** for index configuration, by closely examining the **B-tree index scan process** during the development of the **Index Scan Cost Estimator**. The study also highlights the limitations of the ESR rule, demonstrating that it is not always the optimal solution for index selection.

(2) **Index Scan Cost Estimator**: A key contribution of this thesis is the development of the **Index Scan Cost Estimator**. This estimator introduces a novel algorithm capable of accurately determining the number of index entries to be examined for a given query when the **B-tree** structure is employed. This advancement enables **Index Minder** to distinguish between indexes more effectively, leading to better recommendations. Moreover, this algorithm has the potential to be adapted for other database systems that utilise the **B-tree** structure.

(3) **Development of an Enhanced Index Recommender – Index Minder**: This thesis presents **Index Minder**, an advanced index recommender for **MongoDB**, building on the foundation of **Mindexer**. By integrating the **Index Scan Cost Estimator**, Index Minder overcomes the

limitations of Mindexer, providing a more accurate estimation of theoretical index benefit and significantly improving the quality of recommended indexes.

(4) **Introduction of Conservativeness Control**: A key feature introduced in Index Minder is the **conservativeness control** mechanism, which provides a way to adjust the benefit threshold for recommending indexes. This approach accounts for the impact of update queries by ensuring that only indexes with substantial benefits are recommended, balancing performance improvements with maintenance overhead. Additionally, the conservativeness control offers potential for Index Minder to adapt to varying hardware and software environments, enhancing its flexibility and effectiveness.

(5) **Thorough Performance Evaluation and Comparison**: A comprehensive performance evaluation and comparison between **Mindexer** and **Index Minder** is presented, showcasing the superior performance of Index Minder in recommending high-quality indexes across various workloads. The study employs statistical analysis to ensure robust comparisons, further validating the effectiveness of the proposed enhancements.

1.4 Overview of Thesis Structure

The remainder of this thesis is organised as follows:

- **Chapter 2** provides a comprehensive background on **MongoDB** and indexing strategies, including existing index recommendation tools and their limitations.
- **Chapter 3** presents the **literature review**, summarising related research on index recommendation for both relational and NoSQL databases, and identifying gaps that this thesis addresses.
- **Chapter 4** details the methodology used to design and develop **Index Minder**, focusing on the engineering research approach used in this project. It includes the design of an analysis of flaws in Mindexer, the design of a study on the index structure that led to the new estimator, and the design of an evaluation and comparison of **Index Minder** with **Mindexer**. Additionally, it includes the methodology for assessing and comparing index performance as well as index recommender performance.
- **Chapter 5** presents the experimental results, the new index cost estimation process, a performance evaluation of Index Minder, and a comparative analysis between **Index Minder** and **Mindexer**, highlighting improvements in index recommendation quality and summarising key findings from this study.

- **Chapter 6** discusses the implications of the findings, with an emphasis on the core feature of the **Index Scan Cost Estimator**, which accurately computes the theoretical index scan cost, and the **conservativeness** of **Index Minder**, as well as the additional features introduced. The rationale behind these improvements is also discussed.
- **Chapter 7** addresses the **threats to validity**, discussing the validity of the results obtained in this study.
- **Chapter 8** presents the **limitations and future work** for **Index Minder**, including proposed modifications and next steps to improve the tool.
- **Chapter 9** concludes the thesis, summarising the key contributions and suggesting avenues for future research, including potential adaptations of **Index Minder** for other database systems and improvements to the conservativeness control mechanism.

CHAPTER 2

Background

This chapter provides the necessary foundation for understanding the key concepts, challenges, and methodologies related to MongoDB and index recommendation. Section 2.1 offers an overview of the technical area, introducing MongoDB, NoSQL databases, and the problem domain. Section 2.2 outlines the historical context and identifies research gaps in index recommendation for MongoDB. Section 2.3 presents relevant index recommendation techniques, including manual and automated methods. Finally, Section 2.4 summarises the key terminologies and notations, aiding comprehension of subsequent chapters.

2.1 Overview of the technical area

2.1.1 MongoDB Overview

Relational databases use a table-based data model (Codd, 1970) where entities are represented as rows within a table, and each column corresponds to an attribute of those entities. This model offers a structured and well-defined schema, with each row representing a distinct entity and each column describing specific properties of those entities.

In contrast, MongoDB, a NoSQL database, adopts a document-based data model. MongoDB is a general-purpose document store database in which real-world entities are represented as JSON-like documents (MongoDB, Inc., No date). These documents contain *fields*, which describe the attributes or properties of the entities. Throughout this thesis, the term *field* will be used rather than *attribute*, to maintain consistency with MongoDB's official documentation.

One of the key advantages of NoSQL databases like MongoDB is their ability to handle large data sets with high performance and scalability. Unlike relational database management systems (RDBMSs), which are not typically designed for horizontal scaling, NoSQL databases can distribute data across multiple servers, making them highly scalable and cost-effective for large-scale applications. This scalability is enabled by MongoDB's flexible schema and rich data model, which supports nested data and denormalised storage. For instance, one-to-many relationships can be stored as arrays of nested objects within a single JSON document, allowing data retrieval without the need for costly joins across multiple servers. However, this flexibility also introduces challenges for implementing traditional cost-based index recommenders, as the lack of a rigid schema complicates the index recommendation process. Despite these challenges, NoSQL databases have seen rapid adoption, especially in industries requiring flexible and scalable data solutions.

Data management in MongoDB operates at three hierarchical levels:

(1) **Document Level**: Each document corresponds to an entity and is stored in a BSON (Binary JSON) format (BSON, 2024), offering flexibility in the structure of the data.
(2) **Collection Level**: A collection is a grouping of related documents, analogous to a table in relational databases. However, MongoDB collections are schema-less, allowing each document within a collection to have a different structure.
(3) **Database Level**: The database contains multiple collections, serving as the primary container for all collections and documents. It acts as the central access point for data management within MongoDB.

2.1.2 MongoDB Reads

The basic methods of interacting with a MongoDB server are referred to as **CRUD operations**, which stand for Create, Read, Update, and Delete (MongoDB, Inc., 2024a). In this thesis, the focus is primarily on *read* operations (also referred to as *retrieve* operations), which aim to extract specific data from collections. The terms *read* and *retrieve* will be used interchangeably. These read operations, also known as queries, target specific collections and are composed of *predicates*, which define the conditions used to filter or match documents (analogous to the `WHERE` clause in SQL).

Throughout this thesis, the following notations will be used for clarity:

- **Q**: A query.
- **P**: A predicate, which specifies a condition for filtering documents.

Subscripts will be employed to number queries and predicates. For instance, the notation $Q_1 = \{P_1, P_2, \dots\}$ represents a query Q_1 with predicates P_1 and P_2. A predicate such as $P_1 = \{\text{field}_1 : \text{value}\}$ filters documents where `field_1` equals a specified value, and $P_2 = \{\text{field}_2 : \langle\text{expression}\rangle\}$ allows for conditions such as inequalities or range-based filters (e.g., greater than, less than).

Data retrieval typically involves multiple queries, which can be grouped into a **workload**. A workload, denoted as W, represents a set of queries to be processed by the database. For example, $W = [Q_1, Q_2, \dots]$ describes a workload comprising multiple queries $Q_1, Q_2, \dots$.

2.1.3 MongoDB Indexing

Indexes are specialised data structures that serve as fast lookup guides by storing a subset of the actual data (i.e., a limited number of fields). In MongoDB, indexes are implemented using a **B-tree** data structure, which enables efficient, sorted traversal of the index. A B-tree is a balanced search tree that ensures uniform search costs across all leaf nodes, maintaining balanced access times regardless of the size of the data set (MongoDB Documentation, No date). Each element in the index is referred to as an *index key*.

MongoDB supports both **primary** and **secondary** indexes. The primary index is automatically created on the special field `"_id"` within every document. This field is unique within the collection to which the document belongs, and it is immutable. Any additional indexes that a user creates are classified as **secondary indexes**.

While the primary index only covers the `"_id"` field, **secondary indexes** can be created on any field or combination of fields within a document. When an index is created on multiple fields, it is referred to as a **compound index**. The arrangement and combination of fields within a compound index are described as the *index configuration* and will be denoted in this thesis as {field1: 1, field2: -1, ...}. In a B-tree structure, a compound index is sorted first by the leftmost field, and sorting continues with subsequent fields in the order specified during index creation. The flag **1** indicates ascending order, while **-1** signifies descending order.

The order of fields in a compound index is important for query performance, as it can determine whether the index is usable or unusable for filtering based on predicates or for sorting the results. In this thesis, **indexes** will be denoted by **I**, with subscripts representing the index configuration. For instance, an index created on the field "field1" with ascending order will be denoted as $I_{\text{field1\_1}}$, while a compound index with fields "field1" and "field2" will be represented as $I_{\text{field1\_1\_field2\_1}}$.

2.1.4 Query Execution in MongoDB

MongoDB's **query planner** generates multiple **query execution plans** for each query. The query planner selects the optimal plan based on its ability to produce the most results while performing the least amount of work during the trial period (MongoDB, Inc., 2024e).

MongoDB executes queries in two main ways:

(1) **Collection Scan**: If no index is available, every document in the collection is scanned and evaluated against the query's predicates, which is generally less efficient.
(2) **Index Scan**: If an index exists, MongoDB uses it to quickly locate documents that match the query's predicates, reducing the number of documents that need to be examined and improving performance.

2.2 Historical Context and Research Gaps in MongoDB Indexing

The rise of NoSQL databases, particularly MongoDB, has been driven by the growing need to manage large-scale, unstructured data in a flexible and scalable manner. MongoDB's document-based data model and ability to handle massive amounts of data through horizontal scaling have made it a popular choice for modern applications. However, despite the rapid adoption of MongoDB, the field of **index recommendation** for MongoDB and other NoSQL databases has not evolved at the same pace.

While **indexing** has been a crucial part of relational database systems for decades, providing efficient query performance, the same level of development has not occurred in the NoSQL domain. There is a clear **research gap** in the development of advanced tools for recommending indexes in MongoDB, particularly tools that can account for the unique characteristics of NoSQL data models. Unlike traditional relational databases, where extensive research has been conducted on index selection and optimisation, the index recommendation problem in MongoDB remains underexplored.

2.3 Relevant Index Recommendation Methodologies in MongoDB

2.3.1 Manual Index Recommendation: The ESR Rule

MongoDB provides a widely recommended guideline for constructing compound indexes, known as the **ESR rule** (Equality, Sort, Range) (MongoDB Documentation, 2024). This rule offers guidance to database administrators aiming to manually create effective indexes for optimising query performance.

According to the ESR rule, fields within a query are categorised into three types based on their role in the search:

(1) **Equality Search**: Fields where the query specifies exact matches (e.g., `field = value`).
(2) **Sort**: Fields that determine the order of the query results.
(3) **Range Search**: Fields where the query specifies a range of values (e.g., `field > value` or `field < value`).

The recommended index configuration prioritises **equality fields** first, followed by **sort fields**, and finally **range fields**. The rationale behind this order is to reduce index search cost while maintaining the ability to sort with the index.

The ESR rule, though a general guideline, provides a foundational strategy for enhancing query performance through manual index construction, subject to specific query patterns and workloads.

2.3.2 Atlas Performance Advisor

Atlas Performance Advisor is MongoDB's official index recommendation tool, available exclusively on **MongoDB Atlas**—MongoDB's fully managed cloud database service that simplifies the deployment, operation, and scaling of MongoDB databases. The Atlas Performance Advisor provides index suggestions by analysing the query workload on databases hosted within MongoDB Atlas (MongoDB, Inc., 2024d). However, Atlas Performance Advisor has limitations: it operates only on hosted Atlas databases, not on local MongoDB deployments, and it bases its recommendations solely on query patterns, without considering data distribution.

These limitations can lead to suboptimal recommendations in certain cases, as data distribution can significantly impact query performance. For example, consider an e-commerce database with a field like *paid*, where 99% of entries are *true* and only 1% are *false*. A query for documents where *paid: false* is highly selective and would benefit from an index on the *paid* field. Conversely, a query for *paid: true* would match nearly the entire collection, making an index on *paid* not only unnecessary but potentially detrimental to performance. This example highlights the need for an index recommendation tool that considers both query patterns and data distribution.

2.3.3 Mindexer

To address the limitations of Atlas Performance Advisor, an experimental tool called **Mindexer** was developed as a command-line tool written in Python. Mindexer recommends indexes for MongoDB based on a given query workload and a small random sample of the dataset, thereby incorporating data distribution into its recommendations. By leveraging MongoDB's system profiling, Mindexer analyses query logs to suggest indexes that align with the specific data characteristics and query workload (Rueckstiess, 2023). Despite its innovative approach, Mindexer is an *unsupported* and *experimental tool* that is not officially endorsed by MongoDB Inc. and is recommended only for use in testing environments, rather than production systems.

The motivation for developing Mindexer stemmed from the limitations of Atlas Performance Advisor, particularly the need to incorporate data distribution into index recommendations to improve query performance in diverse scenarios.

2.3.3.1 System Architecture of Mindexer

Mindexer follows a structured process to recommend indexes for MongoDB, consisting of several key components: sampling, workload collection, candidate index generation, benefit estimation, and greedy selection of indexes. The process is illustrated in Figure 2.1, which appears on the following page, and is described in detail below.

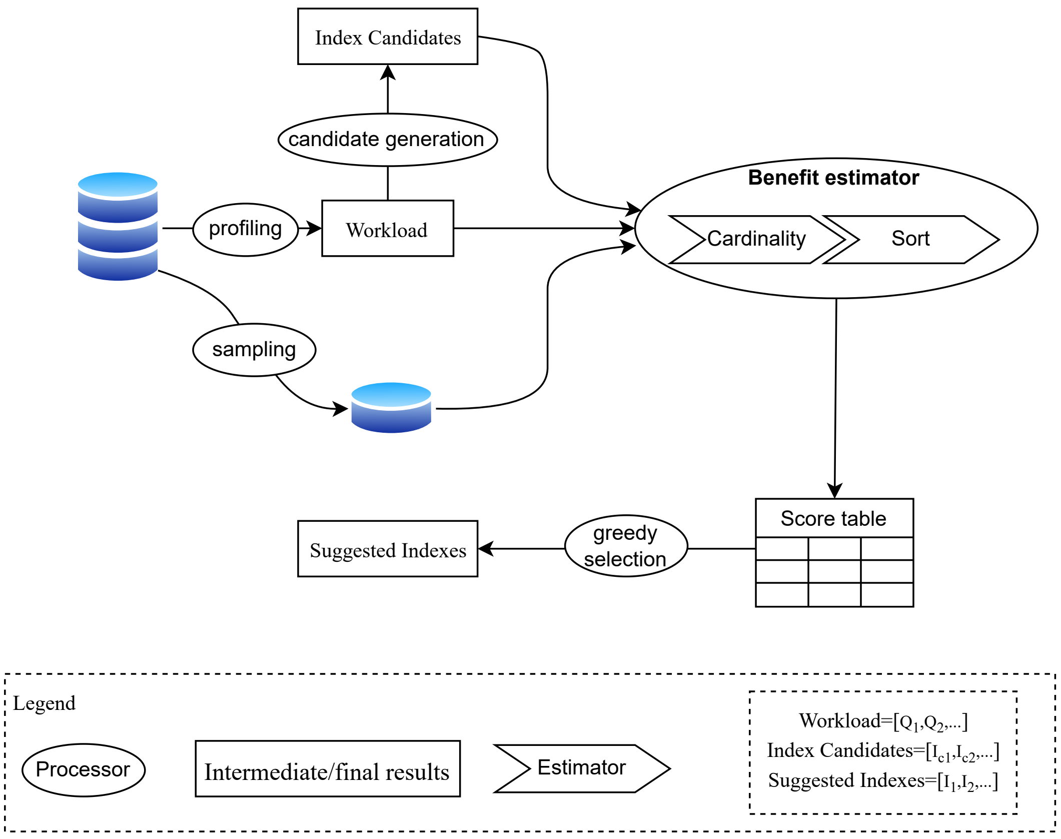

FIGURE 2.1: System Architecture of Mindexer

Sampling from the Database. The system begins by **randomly drawing samples** from the database by using the built-in aggregation stage `$sample`. The sample size is determined by a sample ratio between 0 and 1, with the constraint that at least 1,000 documents must be included to ensure the sample is representative. This sampling step allows Mindexer to reduce the computational cost of cardinality estimation by working with a subset of the database rather than the full collection.

Workload Collection. Next, Mindexer collects the **workload (W)** by making requests to the MongoDB **Database Profiler**, which logs **CRUD operations** as well as configuration and administration commands. However, for the purposes of index recommendation, only CRUD operations are relevant. Each query from the profiler is stored in a list, which represents the workload. This workload is used to determine which indexes could potentially improve query performance (MongoDB, Inc., 2024b).

Candidate Index Generation. Mindexer generates **candidate indexes** by iterating through all possible permutations of up to three fields involved in the queries. Limiting indexes to a maximum of three fields ensures a manageable number of combinations, aiming to reduce the number of documents scanned and improve performance.

Benefit Estimation. Once the workload, sampled data, and candidate indexes are defined, they are passed to the **Benefit Estimator**. The purpose of this component is to evaluate how much **benefit** each index can bring, based on its ability to reduce query costs compared to a **collection scan** (i.e., scanning every document). The benefit is calculated by subtracting the cost of using the index from the cost of performing a full collection scan. The larger the difference, the greater the benefit.

Each index's benefit is stored in a table, where **rows represent queries** and **columns represent candidate indexes**. The value in each cell corresponds to the benefit an index can bring to a specific query. The estimator operates with two sub-components:

- **Cardinality Estimator**: This estimator determines how many documents can be efficiently retrieved using the index for a specific query. Typically, this number is lower than the total number of documents in the collection, making this reduction in examined documents the primary way an index lowers query cost.
- **Sort Estimator**: This estimator assesses whether the index can be used to fulfil a query's sorting requirements (if the query includes a `sort` modifier). If the index supports sorting, the query can avoid in-memory sorting, and the omitted sorting cost is added to the index's benefit.

Greedy Selection of Indexes. Once the benefit table is complete, Mindexer enters the **greedy selection phase**. The total benefit for each index is calculated by summing up the benefits across all queries (i.e., summing each column). The index with the highest total benefit is selected first. After selecting an index, its benefit is **subtracted** from the remaining indexes for each query, and its corresponding column is removed from the table. This process repeats until no index provides a positive summed benefit, ensuring that only the most beneficial indexes are selected.

2.3.3.2 Limitations of Mindexer

While Mindexer offers a starting point for index recommendations, it has notable limitations. One key flaw, identified in a previous thesis, is that Mindexer does not recommend indexes that follow the **ESR**

rule, which is crucial for optimising query performance in MongoDB (Thirukumaran, 2023). This shortcoming, alongside its experimental status, underscores the **lack of robust, production-ready solutions** for index recommendation in MongoDB. Addressing this gap forms the basis of this thesis, which seeks to enhance index recommendation methods for MongoDB by providing more accurate and scalable solutions that adhere to best practices such as the ESR rule.

2.4 Summary of Key Terminologies and Notations

- **Q**: Represents a **query**. Queries are requests to retrieve, insert, update, or delete data from a MongoDB collection.
- **P**: Represents a **predicate**, which is a condition used to filter documents in a query. The notation $P(\text{field})$ will be used to indicate the field that the predicate focuses on.
- $Q_1 = [P_1, P_2, \ldots]$: A notation for a query with multiple predicates, where Q_1 is the query and P_1, P_2 are the predicates.
- **W**: Represents a **workload**, which is a set of multiple queries. For example, $W = [Q_1, Q_2, \ldots]$ denotes a workload consisting of queries $Q_1, Q_2, \ldots$.
- **I**: Represents an **index**.
- $I_{\text{field1\_1\_field2\_-1}}$: A compound index created on two fields, `field1` in ascending order (denoted by 1) and `field2` in descending order (denoted by -1).
- $|D(\text{condition})|$: The number of documents in a collection that satisfy a specified condition, where the condition represents a query or a set of predicates.
- **ESR Rule**: A guideline for constructing compound indexes in MongoDB, where **E** represents equality fields, **S** represents sort fields, and **R** represents range fields.

CHAPTER 3

Literature Review

This literature review is organised around four main topics that are central to the development of an index recommender algorithm for optimising query performance in MongoDB:

(1) Query Optimiser and Query Cost Evaluation
(2) MongoDB Properties
(3) Index Selection Algorithms
(4) Index Selection Algorithm Evaluation

Each paper discussed within these topics will be identified according to the research method it employs. The five key research methods considered in this review are:

(1) Engineering research method
  - Proposes a new design or algorithm.
  - Builds a proof-of-concept system to demonstrate the feasibility of the proposed design.
  - Measures the performance of the new system and compares it to existing solutions.

(2) Mathematical method
  - Defines mathematical terms and establishes relationships between them.
  - Develops and proves theorems based on the defined terms and relationships.

(3) Scientific method
  - Formulates a hypothesis about a phenomenon or relationship between variables.
  - Designs and conducts experiments to collect data relevant to the hypothesis.
  - Analyzes the data to see if it supports or rejects the hypothesis.

(4) Data-driven research approach
  - Collects data from existing sources or through new data gathering methods.
  - Analyzes the data to identify patterns, trends, or relationships.
  - Develops models or explanations that account for the observed patterns in the data.

(5) Descriptive research method
    - Describe a phenomenon, situation, or population accurately and systematically.

By organising the literature around these four key topics and explicitly identifying the research methodology used in each study, this review aims to provide a structured and comprehensive exploration of the field, laying the groundwork for the development of an effective index recommender for MongoDB.

3.1 Query optimiser and query cost evaluation

As a key component of relational databases, the query optimiser plays a crucial role in query cost evaluation and query execution plan generation. Since cost evaluation significantly impacts index selection during query plan creation, this section delves into the characteristics of query optimisers and their influence on designing effective index selection algorithms.

3.1.1 Query optimiser

When a DBMS receives a query, the database engine needs to generate a valid query execution plan. There are generally infinitely many possible execution plans for the same query. Finding the optimal plan is crucial for efficient query processing. This process, called query optimization, is performed by the query optimiser. A classic approach is the cost-based approach, which enumerates possible plans and uses cardinality estimation to feed a cost model. This model then chooses the plan with the lowest estimated cost (Leis et al., 2015). The paper *How Good Are Query Optimizers, Really?* highlights the lack of systematic testing for the performance of query optimisers on multi-join queries.

This paper makes several key contributions:

- Design of the Join Order Benchmark (JOB).
- Completion of the first end-to-end study of the join ordering problem using a real-world dataset and realistic queries.
- Analysis of the influence of cardinality estimation, cost model, and plan space enumeration on the query optimization process.

These contributions are demonstrated by running experiments with JOB against PostgreSQL with controlled variables to gather quantitative evidence. The research method used is scientific method. The authors' findings, based on this evidence, are:

- Cardinality estimation heavily impacts optimiser performance. Misestimates can lead to suboptimal, even catastrophic, query execution plans.
- The type and number of indexes significantly influence the plan search space and, consequently, the system's sensitivity to cardinality misestimates.

This paper is indirectly related to the research topic. Query optimization is essential for query processing. The optimiser evaluates indexes and determines which ones to use for query execution. Understanding the theory behind index selection by the optimiser is crucial before recommending indexes.

My research focuses on recommending indexes in NoSQL databases. The query optimiser in a NoSQL database might have entirely different characteristics compared to a relational database. Adapting the experiment from this paper could be beneficial to examine the characteristics of NoSQL databases.

3.1.2 Use of the Query optimiser

Database design tools rely on query optimisers to compare different physical design options. While optimisers offer accurate cost models, they are computationally expensive, slowing down design tools, especially for large datasets (Papadomanolakis et al., 2007). *Efficient Use of the Query Optimizer for Automated Physical Design* addresses this issue by introducing INUM (INdex Usage Model).

INUM is a key contribution in this paper, which tackles the trade-off between accuracy and speed in cost estimation. Unlike optimisers, INUM delivers cost estimates significantly faster (up to three times faster). This dramatically improves the running time of index selection algorithms. Crucially, INUM maintains the same level of accuracy as the query optimiser, ensuring reliable cost estimations for selecting optimal index configurations.

The paper demonstrates INUM's effectiveness through various means. It details the framework's core principles and how it achieves high speed without sacrificing accuracy. It also showcases integration with existing index selection algorithms, significantly boosting their speed. Additionally, real-world evaluations show substantial performance improvements in a commercial query optimiser. Finally, the paper highlights how INUM allows for the evaluation of more candidate indexes, leading to the selection of better index configurations. This research method is engineering research method.

While INUM is designed for SQL databases, limiting its direct application to other database systems, this work is indirectly related to the research topic. However, ensuring consistency between index performance estimation in index selection and database query optimization remains crucial for recommending useful indexes.

3.1.3 Predicate selectivity estimation

Cardinality estimation is a crucial step in cost-based query optimization. Misestimating cardinality can lead to choosing a sub-optimal query execution plan, which can significantly impact performance (reference). Estimating predicate selectivities is one aspect of cardinality estimation. Traditional database systems use internal functions to compute these selectivities, and this area has been well-studied (Chawathe, 2020). However, such estimates haven't been adequately studied in the context of NoSQL database systems, such as cloud-based database services.

The paper *Estimating Predicate Selectivities in a NoSQL Database Service* identifies this gap in selectivity estimation research. The paper's key contributions are: formulating the selectivity estimation problem for NoSQL databases by adapting it from traditional databases, designing a sample-based selectivity estimation method with its limitations outlined, and conducting experiments on real datasets to evaluate the method. This research method used here is engineering research method.

The authors use quantitative evidence (data gathered from experiments) to evaluate their contribution. The sampling-based method shows significantly higher accuracy than its theoretical guarantees, making it a promising option for selectivity estimation. Additionally, the evaluation highlights the difference in how NoSQL database systems handle queries compared to traditional databases.

This paper is indirectly related to the research topic. Cardinality estimation, or more specifically, predicate selectivity estimation, plays a crucial role in query optimization and index selection. Accurate estimation is likely the key to efficient query optimization and index selection. This paper proposes a sample-based estimation approach with a defined threshold, providing valuable guidelines.

Considering a sampling approach for index performance evaluation through cardinality estimation could be beneficial, especially when there's no good heuristic available for index performance. This approach can save computational time.

3.2 MongoDB property

This section will first analyse the key properties of indexes. Subsequently, it will explore the relevant properties of MongoDB's data model and query processing.

3.2.1 Index interactions

Database administrators rely on materialized indexes to optimize workload performance. However, existing advisor tools in commercial DBMS recommend index sets without considering interactions between individual indexes (Gupta et al., 1997). These interactions can significantly affect the overall benefit of each index, making materialization decisions challenging for administrators (Schnaitter et al., 2009).

Index interactions in physical design tuning: modeling, analysis, and applications addresses this gap by proposing a framework and tools for analysing index interactions. They formalize the concept of interactions and develop a novel algorithm to identify them within a recommended set. This knowledge is then leveraged in two new database tuning tools:

- Visualisation Tool: This tool helps administrators understand interactions by visually representing a partitioned view of the index set, where partitions contain non-interacting indexes.
- Materialization Scheduler: This tool uses interaction information to compute a schedule for materializing indexes across maintenance windows, maximizing overall benefit.

The paper demonstrates the efficiency of their algorithm and the effectiveness of the tools through experiments on IBM DB2 and strong analytical results. This research method used here is engineering research method. They also outline promising future work, including extending the methodology to analyse interactions with materialized views and exploring incremental characterization for online tuning scenarios.

While this paper focuses on index interactions in relational databases, which is indirectly related to the research topic, exploring these interactions in NoSQL databases is a worthwhile area of investigation. If interactions exist between NoSQL indexes, then independent evaluation of each index configuration during selection may not be sufficient.

3.2.2 Comparing NoSQL MongoDB to an SQL DB

The big data era has ushered in an age of ever-increasing volumes of unstructured data. Traditional database systems, designed for a limited amount of structured data, struggled to keep pace. NoSQL databases emerged as a solution, offering scalability and flexibility for handling massive amounts of unstructured data. As a result, NoSQL solutions are gaining significant traction in a world still largely dominated by SQL relational databases (Parker et al., 2013).

Comparing NoSQL MongoDB to an SQL DB highlights a research gap: the advantages (or disadvantages) of using NoSQL for structured data, which is not necessarily "Big". Existing studies often focus on big data, leaving the performance of NoSQL for smaller structured datasets unclear.

This paper's key contribution is designing and implementing an experiment to compare the runtime performance of a traditional SQL database and MongoDB for a modestly-sized structured dataset. The authors demonstrate their findings through a quantitative analysis of the experiment results. This is scientific research method. Overall, MongoDB exhibits faster runtime performance for inserts, updates, and simple queries compared to SQL. However, SQL outperforms MongoDB when updating and querying non-key attributes, as well as for aggregate queries.

This paper directly aligns with the research topic by revealing performance characteristics of MongoDB, a NoSQL database. This work could be useful when porting index selection algorithms from relational databases to NoSQL systems like MongoDB. Moreover, one finding suggests that indexes in MongoDB might be particularly beneficial for improving the performance of aggregate queries.

3.2.3 Secondary Indexing Techniques in LSM-based NoSQL Databases

A Comparative Study of Secondary Indexing Techniques in LSM-based NoSQL Databases tackles a key gap in NoSQL secondary indexing research. While NoSQL excels in writes and primary key lookups, querying other attributes is crucial for many applications. Existing NoSQL solutions offer various secondary indexes, but research lacks a unified view and comparative analysis (Qader et al., 2018).

This paper's key contributions are:

- Taxonomy Development: proposing a taxonomy for NoSQL secondary indexes, categorizing them as Embedded (lightweight filters within the main table) or Stand-Alone (separate data structures).
- Performance Evaluation: building LevelDB++, a system based on LevelDB that implements two Embedded and three Stand-Alone indexes, representing common approaches in NoSQL databases.

These contributions are evaluated by experimentation and analysis (scientific research method), the paper reveals there's no single dominant technique. Embedded indexes boast superior write throughput and space efficiency, while Stand-Alone indexes deliver faster query response times. This highlights the importance of considering application workloads – write-heavy applications might favor Embedded indexes, while query-intensive ones might benefit from Stand-Alone options. The paper serves as an empirical guide for selecting appropriate secondary indexes in NoSQL environments.

While not directly related to the research topic, this paper explores properties of secondary indexes in MongoDB. Notably, MongoDB utilizes B+ trees for stand-alone secondary indexes, mirroring the approach used by traditional RDBMSs. Consequently, MongoDB secondary indexes inherit the characteristics of stand-alone indexes.

3.3 Index selection algorithm

This section will review existing index recommender algorithms in relational databases, exploring potential adaptations for MongoDB.

3.3.1 Index selection for OLAP

Data warehouses are often used by enterprises. They act as central repositories for integrated data from various disparate sources. These warehouses typically store massive amounts of data, and Online Analytical Processing (OLAP) queries are used to extract insights from it. The data analysed through OLAP is called a "data cube," indicating its multidimensional nature. Due to the complexity of OLAP queries and the big data nature of data warehouses, most systems typically precompute summary tables of the original data and build indexes on those summaries (Gupta et al., 1997). *Index Selection for*

OLAP has pointed out a common issue exists in most commercial OLAP systems: the separation of precomputing summary tables and building indexes. This two-step process can perform poorly due to the inefficient use of space when using a trial-and-error approach for allocating space between summary tables and indexes.

The key contributions of this paper are the followings. The authors propose a new approach that considers summary tables and their indexes in the same stage. They first formulate the problem of suggesting optimal indexes for a set of OLAP queries with space constraints as a bipartite multigraph problem. The optimal solution is then defined as a set of indexes and a summary table that offer the lowest cost for targeted queries. An r-greedy algorithm is used to search for the optimal solution, allowing for adjustments in the complexity of the index searching process.

The contributions are evaluated by compare the performance between two-step approach and one-step approach against TPC-D benchmark. The research method used here is engineering research method. The results show that integrating the precomputation of summary tables and indexes into one step improves index recommendation performance. Additionally, there is a negligible performance gap between an algorithm of moderate complexity and one of high complexity. The authors demonstrates a common problem formulation of searching indexes and a cost-based approach for selecting indexes, offering accurate performance comparisons, but potentially leading to longer verification times. They also considered space constraints, making index selection more challenging but reflecting real-world limitations. The use of the r-greedy algorithm plays a crucial role in achieving a trade-off between algorithm complexity and performance.

This work is indirectly related to the research topic. Since data warehouses experience a significantly higher frequency of select queries compared to update queries, the authors' decision to disregard the potential increase in update cost caused by index candidates is reasonable. However, my research topics involves general-purpose database systems like MongoDB, the impact on update query costs should not be ignored.

The use of a complexity-adjustable search algorithm is inspiring. In my future work, I might adapt this approach to achieve a trade-off between complexity and performance.

3.3.2 Index selection in relational databases

In relational databases, there are two main types of indexes: primary and secondary. The paper *Index Selection in Relational Databases* highlights an issue: studies on index selection often focus primarily on secondary indexes. To address this, the authors designed a study that examines the selection of both primary and secondary indexes, along with their relationship. To reduce complexity, the general selection problem was divided into separate problems for primary and secondary indexes (Choenni et al., 1993).

This paper makes key contributions:

- Formulating a general index selection problem that considers both primary and secondary indexes.
- Exploring the relationship between primary and secondary indexes, particularly the order of selection.

The authors evaluate these contributions using an analytical approach (scientific research method applied). They developed a general cost model for index selection that incorporates database statistics and considers both query and maintenance costs, unlike previous models (Gupta et al., 1997). By applying this method to both selecting the primary index before secondary indexes and vice versa, the authors demonstrate the dependency between primary and secondary index selection. The interaction between indexes were examined further by (Schnaitter et al., 2009). They conclude that selecting primary indexes first is generally better, but careful consideration is needed to avoid sub-optimal configurations.

This paper is indirectly related to the research topic. It reveals the existence of dependencies between indexes during selection. Future research on index selection should consider the order and dependencies between different index types.

Since MongoDB is not a relational database, its indexes might behave differently. Exploring the relationship between indexes in MongoDB could be helpful before recommending indexes in general.

3.3.3 An efficient cost-driven index selection tool for Microsoft SQL server

Traditional database administration involves significant manual effort, especially for large-scale deployments. To address this challenge, *An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL*

Server investigated techniques for self-tuning databases. One crucial aspect of database performance is physical design, which includes selecting appropriate indexes (Chaudhuri and Narasayya, 1997). This paper addresses limitations in current index selection algorithms, such as scalability and neglecting the relationships between indexes (Gupta et al., 1997). To bridge this gap, it proposes novel techniques employed in AutoAdmin for automating index selection.

The key contribution of the paper is the novel index search technique. It employs a unique search method that filters out irrelevant indexes early and leverages the relational database engine's features (using minimum optimiser calls to determine query cost) to minimize search cost. Additionally, it explores options iteratively (eliminating index with low benifit), building complex indexes (like multi-column ones) from promising simple ones (like single-column).

These contributions are evaluated by the implementation of novel search technique and test results against TPC-D (engineering research method applied). The tool is fully functional on Microsoft SQL Server 7.0 (with some server modifications). Experiments show significant improvements (4-10 times faster) in search time without compromising the quality of the chosen indexes. These results support the effectiveness of the proposed strategies.

This paper, although indirectly related to the research topic, offers valuable practical approaches for achieving scalability in index recommenders. Reducing algorithm complexity is crucial for efficiency gains. Filtering out irrelevant indexes dramatically reduces the candidate search space, while evaluating index performance using the database's built-in optimiser ensures the recommended indexes actually improve query processing speed.

In MongoDB project, adapting the technique of filtering out irrelevant indexes could significantly reduce the number of candidate indexes, leading to better algorithm efficiency. However, the MongoDB optimiser needs careful examination to ensure the index selection algorithm aligns with the optimiser's behaviour.

3.3.4 Index Selection for Databases: A Hardness Study and a Principled Heuristic Solution

In real-world scenarios, selecting indexes often involves user-specified storage constraints. The paper *Index Selection for Databases: A Hardness Study and a Principled Heuristic Solution* formulates the index selection problem as recommending a set of indexes that maximize benefit within a given workload and storage constraint. The authors highlight a research gap in the hardness of the index selection problem and propose a novel approach (Chaudhuri et al., 2004).

The first key contribution is formalizing the index selection problem. The authors demonstrate this by converting the problem into a mathematical representation that incorporates constraints.

The second key contribution is a Hardness Study on the index selection problem. This contribution is evaluated through dividing the problem into two parts: clustered and non-clustered index selection. By converting each part into the k-densest subgraph problem and the MAX Constraint Satisfaction Problem, the authors conclude that the index selection problem is NP-hard.

The first and second contribution are made by applying mathematical research method.

The third key contribution is the development of a new algorithm for the problem, treating it as a knapsack problem. This contribution is demonstrated by applying linear programming to assign benefits to indexes and choosing the index with the highest benefit in each iteration. Unlike the low-benefit elimination approach and optimiser cost evaluation approach mentioned in (Chaudhuri and Narasayya, 1997), this algorithm greedily pick index with highest benefit and explicitly defines a cost model. This contribution is evaluated by experimenting with the algorithm against benchmarks and synthetic datasets (engineering research method). The results show that the novel algorithm is more scalable while achieving similar quality.

The paper is indirectly related to the research topic as it proposes a different approach for index selection. It proves that solving the index selection problem is generally NP-hard, implying that evaluating index performance by actually running them with the entire workload might not be feasible for scalability reasons. In my MongoDB project, considering heuristics can be beneficial for examining index performance and selection.

3.3.5 Learned Index Benefits: Machine Learning Based Index Performance Estimation

The optimiser plays an important role in choosing the optimal query execution plan. This process relies on the core functionality of the optimiser: estimation of query cost (Leis et al., 2015). In traditional database systems, like SQL, there is a "what-if" API. However, the limitations of the optimiser's "what-if" functionality lead to significant errors in index recommendation based on it. Additionally, using the "what-if" API to estimate the cost of each index configuration takes a significant amount of time (Shi et al., 2022).

The paper "Learned Index Benefits: Machine Learning Based Index Performance Estimation" highlights the gap in the study of Machine Learning Based Index Performance Estimation due to these limitations.

The key contribution of this paper is proposing an effective end-to-end machine learning-based index benefit estimator (LIB), which is the first method to use machine learning to estimate index benefit for index selection.

This contribution is evaluated by experimenting with their new approach against four benchmarks: TPC-H, TPC-DS-10, TPC-DS-50, and IMDB-JOB (engineering research method applied). The results indicate that LIB outperforms "what-if" based index benefit estimations in terms of accuracy and efficiency. This is likely because machine learning algorithms can more accurately capture the characteristics of the data and workload. Additionally, transfer learning is employed to enhance adaptation to new database schemas.

This paper is directly related to the research topic. It introduces machine learning as a tool for index selection and demonstrates a promising direction for future work. The paper also serves as a reminder that built-in database features might become outdated, and their accuracy should be evaluated. However, it's important to remember that the accuracy of a machine learning model heavily relies on the training data, so caution is necessary during the training process.

3.3.6 How artificial intelligence (AI) can benefit automated indexing (AI)

Automated database indexing aims to optimize query performance by recommending suitable index configurations. However, a major hurdle lies in the inaccuracy of cost estimates provided by traditional query optimisers. These estimates often lead to index recommendations that degrade real-world query execution times (Ding et al., 2019).

AI Meets AI: Leveraging Query Executions to Improve Index Recommendations tackles this gap by proposing a novel approach that leverages machine learning (ML) for improved accuracy in automated index tuning. The core insight is to move away from relying on optimiser estimates. Instead, the paper proposes formulating the task of comparing execution costs between different index configurations as a classification problem for an ML model (key contritbution). This approach demonstrably offers significantly higher accuracy in cost comparisons.

The paper highlights the ease of implementing this new approach. The proposed ML classifier can be integrated seamlessly into existing index tuners with minimal modifications. This allows for a smooth transition from traditional estimates to AI-powered indexing.

The paper's contribution is validated through a comprehensive evaluation process (engineering research method applied). It explores the design space for implementing the ML classification approach. Additionally, the effectiveness of the model is demonstrated using industry-standard benchmarks and real customer workloads. Results show up to a 5x reduction in errors when identifying the more efficient execution plan. This translates to virtually eliminating query performance regressions when the model is used within index tuners.

This paper directly aligns with the research topic. By bypassing the reliance on relational database APIs (like query optimisers) for index performance estimation, this approach presents a potential adaptation for the index selection problem in NoSQL database systems.

3.4 Index selection algorithm evaluation

Evaluating index selection algorithms requires a systematic framework and industry-standard benchmarks. This section examines existing evaluation practices and explores relevant industry benchmarks for databases.

3.4.1 Index selection algorithm evaluation

This *Magic mirror in my hand, which is the best in the land?: an experimental evaluation of index selection algorithms* addresses a critical challenge in database optimization – selecting the most effective indexes for a given workload. While various index selection algorithms exist, their strengths and weaknesses remain unclear, making informed selection difficult (Kossmann et al., 2020).

Existing research offers a multitude of index selection algorithms, ranging from simple heuristics to complex optimization approaches. However, a comprehensive comparison of these algorithms across various performance metrics is lacking, that is a gap in research this paper identified.

This paper bridges the gap by performing a thorough comparative analysis of eight prominent index selection algorithms. The key contributions are:

- Comparative Analysis Framework: The paper establishes a comprehensive framework for evaluating index selection algorithms. This framework considers factors such as solution quality (effectiveness of chosen indexes), runtime efficiency, support for multi-column indexes, and solution granularity (individual indexes vs. index groups).
- Algorithm Evaluation and Recommendations: The paper evaluates the eight algorithms using established benchmarks (Join Order (Leis et al., 2015), TPC-H, and TPC-DS) and an extensible evaluation platform for reproducibility. Based on the evaluation, the paper provides specific recommendations for choosing the best algorithm in various scenarios (e.g., budget constraints, workload size).

The paper demonstrates its contribution through extensive experimentation on the PostgreSQL database system (engineering research method applied). The evaluation platform facilitates exploration of different scenarios and user needs. The paper concludes by highlighting the strengths and weaknesses of each algorithm, leading to recommendations for their optimal usage. Additionally, the platform paves the way for future research on improved solution concepts, such as hybrid approaches that leverage the strengths of different algorithms. It is important to exercise caution when interpreting these results. The evaluation relies heavily on the query optimiser, and the accuracy of its cost estimations can be unreliable (Leis et al., 2015) (Papadomanolakis et al., 2007).

This paper offers a valuable framework for comparing index recommendation algorithms. It highlights the importance of considering not only the quality of recommended indexes but also factors like runtime efficiency, support for multi-column indexes, granularity (individual vs. grouped indexes), and algorithm complexity.

While this paper is indirectly related to the research topic, the framework for evaluating index recommendation algorithms could potentially be adapted to NoSQL databases. However, it's important to note that the cost estimation relies on the PostgreSQL optimiser, which wouldn't be directly transferable to other database systems.

3.4.2 TPC database benchmark

Keeping the TPC relevant! discusses the historical and ongoing importance of industry-standard benchmarks (TPC) for evaluating and comparing database systems. However, the paper identifies a gap – the need for TPC benchmarks to adapt to the evolving technological landscape (Nambiar and Poess, 2013).

Traditionally, TPC benchmarks have ensured fair comparisons and driven performance improvements. However, challenges have emerged:

- Slower Development Cycles: New benchmark development cycles struggle to keep pace with rapid innovation.
- Shifting Landscape: Cloud computing, big data analytics, and vendor consolidation impact the relevance and adoption of traditional benchmarks.
- Increased Complexity: Benchmark complexity makes them expensive and difficult to develop and run.

The key contribution of this paper is giving a brief guide of choosing database bench mark and arising an attention to the issues might occur in new benchmarks.

It paper applied descriptive research method, it emphasizes TPC's commitment to adapt and evolve its benchmarking standards to stay relevant in the face of rapid technological advancements.

Since this paper introduces each TPC database benchmark, it directly aligns with the research topic on selecting appropriate benchmarks for effective performance comparisons.

3.4.3 Social networks database benchmark

Database researchers and practitioners rely heavily on benchmarks to compare the performance of different database systems (Armstrong et al., 2013). However, existing benchmarks often struggle to accurately reflect the unique workloads of large-scale web services like social networks.

LinkBench: a database benchmark based on the Facebook social graph addresses this gap by introducing LinkBench, a novel synthetic benchmark specifically designed to mimic real-world social network workloads. LinkBench leverages valuable insights gleaned from analyzing production databases at Facebook, a major social networking platform.

The paper demonstrates LinkBench's effectiveness through a multi-step approach. First, the authors precisely analyse social network data and query workloads at Facebook. This analysis identifies key statistical distributions and highlights the prevalence of power laws in various aspects of the workload (data-driven research approach applied). Based on these insights, LinkBench is accurately crafted to replicate the core aspects of a social network workload. Finally, the paper showcases LinkBench's utility by presenting a performance profile of the MySQL database system under this social network-inspired workload.

This is directly related to the research topic. Evaluating index recommending algorithm cannot be done with out bench mark or synthetic data. LinkBench's open-source availability allows researchers, developers, and database administrators to leverage it for profiling and evaluating various database systems.

In this project, the benchmark will be carefully selected to facilitate the evaluation of the index selection algorithm.

CHAPTER 4

Methodology

This chapter outlines the methodology employed to enhance index recommendations in MongoDB by addressing the limitations identified in the existing tool, **Mindexer**. The research follows a structured engineering research approach, comprising three key phases: (1) performance analysis of **Mindexer** to identify its shortcomings, (2) proposal and implementation of a new index cost estimation method to overcome these limitations, leading to **Index Minder**, and (3) performance evaluation of the enhanced tool, **Index Minder**, along with a comparative analysis between **Index Minder** and **Mindexer**. Recognising the frequent need to assess index performance throughout the research, a dedicated subsection detailing the index performance assessment methodology is included at the beginning of this chapter, followed by a section on index recommender performance assessment and comparison. This methodology is grounded in a quantitative, analytical framework, ensuring a rigorous examination of the research problem and the proposed solution (Deb et al., 2019)[p. 5].

4.1 Index Performance Assessment Methodology

Assessing and comparing index performance is essential for understanding their impact on query execution and for making better index recommendations in MongoDB. Throughout this research—in Phases 1, 2, and 3—index performance assessments are conducted. To avoid redundancy and ensure consistency, this section explains the methodology used for performance assessment before detailing the specific phases of the research.

4.1.1 General Process of Assessing Index Performance

The primary purpose of using an index is to expedite data retrieval, resulting in shorter execution times for read queries. Therefore, the performance of an index or a set of indexes is directly evaluated by comparing the measured execution time of a query with the index(es) and without any secondary indexes.

Since the performance of indexes may vary between workloads, the assessment is always conducted for a specific workload.

For a given workload W (which can consist of one or more read queries) and a set of indexes (which could be a single index or a set of indexes, possibly recommended by an index recommender tool), the general index performance assessment process involves the following steps:

(1) **Execute Workload Without Any Secondary Indexes**:
- Execute the workload W without any secondary indexes (apart from the default `_id` index).
- Record the total execution time of the workload.

(2) **Execute Workload With Index(es)**:
- Create the index(es) under evaluation.
- Run the same workload W with the index(es) in place.
- Record the total execution time of the workload.

(3) **Compare Execution Times**:
- Analyse the difference in execution times to determine the effectiveness of the index(es) in reducing workload execution time.

This process provides a direct comparison of query execution times with and without the index(es), enabling an evaluation of their quality and impact on the workload. A high-quality index is expected to significantly reduce the execution time of the workload; the greater the reduction, the better the index quality.

4.1.2 Execution Time Measurement

4.1.2.1 Mitigating Fluctuations in Execution Time Measurements

Measuring query execution time can be challenging due to fluctuations caused by various factors in computer systems, such as memory usage, CPU load, and system processes. These factors can introduce noise into the measurements, making it difficult to obtain consistent results.

Additionally, the first execution of a query often takes significantly longer than subsequent executions. This initial delay can be attributed to the DBMS compiling the query, developing the execution plan, and loading indexes into memory. To mitigate these fluctuations and achieve more reliable measurements, the following steps are implemented:

(1) **Multiple Executions**: Each query is executed a total of **eleven times**. Multiple executions help in smoothing out anomalies and capturing a representative performance metric.
(2) **Discarding the First Execution**: The execution time from the first run is discarded to eliminate the impact of initial setup delays.

4.1.2.2 Utilising MongoDB's `explain()` Function

To ensure that the measured execution times accurately reflect the query performance without external overheads, MongoDB's `explain()` function is used, specifically the `executionStats` section. This approach offers several advantages:

- **Excludes Client-Side Overheads**: By using `explain()`, the execution time measured pertains solely to the query execution within MongoDB, excluding additional time that might be incurred by the client application (e.g., the time taken by PyMongo to establish a connection to the DBMS).
- **Detailed Execution Metrics**: The `executionStats` section provides in-depth metrics about the query execution, including execution time in milliseconds, the number of documents examined, and index usage statistics.

For a detailed overview of the specific information extracted from `explain()` and guidelines on interpreting its output, please refer to Chapter 10 Appendix.

4.1.2.3 Controlled Measurement Environment

To ensure fairness and consistency in the performance comparison of index(es) against a specific workload, the following control measures are implemented:

- **Consecutive Measurements**: For each workload, all performance measurements—without any secondary index and with a set of indexes under evaluation—are conducted consecutively in a short time period. This approach minimises the impact of time-varying system factors such as background processes or changes in system load.

- **Identical Conditions**: No other actions or operations are performed on the system during the measurement period for a specific query. This isolation helps in maintaining a consistent environment for all measurements within the round.
- **System Monitoring**: System resources such as CPU usage and memory consumption are monitored to ensure that there are no significant fluctuations that could affect the measurements.

4.1.3 Comparing Execution Times

Given that multiple execution times are recorded for each comparison condition (scenario), appropriate statistical methods are used to compare the performance effectively.

4.1.3.1 Using Averages

For an initial comparison:

- **Calculate Average Execution Time**:
  - Compute the mean of the ten execution times for each scenario.
- **Performance Improvement**:
  - Calculate the percentage improvement in execution time when using the index(es) compared to without indexes.

While averages provide a general sense of performance differences, they may not account for variability in the data.

4.1.3.2 Statistical Tests

To assess whether observed differences in execution times are statistically significant:

- **Two-Sample t-Test**:
  - When comparing two sets of execution times (e.g., without indexes vs. with recommended index(es)), a two-sample t-test is used.
  - This test determines if there is a statistically significant difference between the means of the two samples.
- **ANOVA (Analysis of Variance)**:

  - When comparing more than two sets of execution times (e.g., with several sets of index(es)), ANOVA is employed.
  - ANOVA assesses whether there are any statistically significant differences among the group means.
- **Assumptions and Validation**:
  - Verify that the data meet the assumptions of the statistical tests (e.g., normality, homogeneity of variances).
  - The time measurements meet the assumptions of the statistical tests as they originate from identical sources and are expected to follow the same distribution. If assumptions are violated, consider non-parametric alternatives such as the Mann-Whitney U test or Kruskal-Wallis test.
- **Significance Level**:
  - A significance level (e.g., $\alpha = 0.05$) is chosen to determine statistical significance.

Using statistical tests strengthens the analysis by providing evidence that performance differences are not due to random chance.

4.1.4 Measurement Procedure Summary

The overall procedure for assessing index performance against a workload W is as follows:

(1) **Prepare the Environment**:
  - Ensure the MongoDB server is running and the targeted collection is present.
  - Confirm that no other intensive processes are running on the system.

(2) **Execute Workload Without Secondary Indexes**:
  - Run the workload W using the `explain()` function eleven times.
  - Discard the execution time from the first run.
  - Collect the execution times from the remaining ten runs.

(3) **Create Index(es)**:
  - Drop all secondary indexes.
  - Create all index(es) under evaluation.
  - Allow the index creation process to complete before proceeding.

(4) **Execute Workload With the Index(es)**:

- Run the workload W using the `explain()` function eleven times.
- Discard the execution time from the first run.
- Collect the execution times from the remaining ten runs.

(5) **Compare Execution Times**:

- Use averages and/or statistical tests (e.g., two-sample t-test) to analyse the differences in execution times.
- Determine whether the index(es) provide a statistically significant improvement in query performance.

(6) **Repeat for Additional Index Sets (if applicable)**:

- If comparing multiple index recommender tools or index sets, repeat steps 3 to 5 for each set.

4.1.5 Rationale Behind the Methodology

The chosen methodology addresses common challenges in performance measurement and comparison:

- **Mitigating Fluctuations**:
  - Multiple executions and statistical analysis help minimise the impact of transient system states and anomalies.
- **Excluding External Overheads**:
  - Using `explain()` focuses on the database's execution time, providing a more precise assessment of the index's impact.
- **Ensuring Fair Comparison**:
  - Conducting executions consecutively and under identical conditions ensures that all scenarios are subject to the same environmental factors.
- **Statistical Significance**:
  - Employing statistical tests strengthens the validity of conclusions by demonstrating that observed differences are unlikely due to chance.

4.2 Index Recommender Performance Assessment Methodology

The performance of an index recommender is assessed based on the quality of the indexes it recommends for a given workload. The methodology involves measuring and comparing workload execution times, where lower execution times indicate higher-quality indexes and, by extension, a more effective index recommender.

The assessment process begins of the following steps:

(1) Execute the workload without any secondary indexes and record the execution time as a baseline.
(2) Use the index recommender to generate a set of recommended indexes.
(3) Create the recommended indexes and allow sufficient time for their creation to complete.
(4) Execute the workload again with the recommended indexes and record the execution time.

This process is repeated for each index recommender being evaluated, enabling a comparative analysis. The recommended indexes that yield the shortest workload execution times are considered the highest quality, thus indicating the effectiveness of the corresponding index recommender.

Time measurement and comparison follow the same procedures outlined in Section 4.1.

4.3 Engineering Research Approach

The methodology employed in this project follows an engineering research approach, which emphasises practical solutions, iterative development, and real-world applications (Deb et al., 2019). This approach is particularly suitable for the project, as it aims to optimise an existing tool—Mindexer—by proposing modifications that address its limitations.

The process is structured into the following phases:

(1) **Phase 1: Performance Analysis of Mindexer**
(2) **Phase 2: Proposal and Development of Index Minder's Enhanced Cost Estimation**
(3) **Phase 3: Performance Evaluation and Comparative Analysis**

These phases are iterative, allowing for continuous refinement and improvement of the proposed solution. Each phase builds upon the findings of the previous one, ensuring a comprehensive and methodical enhancement of the index recommendation tool.

4.4 Phase 1: Performance Analysis of Mindexer

Mindexer, while effective in many scenarios, exhibits significant limitations. A primary issue is its inability to differentiate between index field orders in compound indexes, which is crucial for query performance, thus failing to adhere to the ESR rule (general best practice) when recommending indexes (Thirukumaran, 2023). To address these deficiencies, a comprehensive performance analysis, including experiments and code review, was conducted to identify its weaknesses.

4.4.1 Experimental Design

The performance analysis of Mindexer commenced with the design and generation of a representative synthetic dataset and workloads.

4.4.1.1 Synthetic Collection Generation

To evaluate the effectiveness of Mindexer, a synthetic dataset was generated to mimic a university use case focused on student information. Each field in the dataset was constructed to follow specific distributions, such as uniform or normal, to capture a range of typical data patterns found in practice. Additionally, the dataset encompassed diverse data types, including numeric and string attributes.

The fields were also varied in size to closely reflect the heterogeneity observed in real-world database applications. This design aimed to capture variability that would enable the investigation of Mindexer's ability to provide accurate index recommendations under diverse data conditions.

Based on the principles mentioned above, a synthetic dataset comprising 100,000 records was created. The collection size is approximately 1.5 GB, which prevents the entire dataset from being loaded into memory and simulates a realistic large-scale database environment. Each record includes the following fields:

- **ID (integer)**: A unique identifier for each record.
- **Name (string)**: Generated using the Faker Python package to simulate realistic names. Approximately 70% of the names are distinct, introducing diversity into the dataset.
- **Age (integer)**: Values generated from a normal distribution with a mean of 23 and a standard deviation of 2, reflecting realistic variability in ages.

- **Mark (float)**: Numerical scores generated from a normal distribution with a mean of 75 and a standard deviation of 4, mimicking a natural distribution of marks.
- **Major (string)**: Randomly assigned from a set of 113 unique majors using a uniform distribution, ensuring equal probability for each major.
- **Description (string)**: A long string added to each document to increase the document size. This field serves as payload to enlarge the size of the collection and is crucial for simulating scenarios where data cannot be entirely held in memory. No predicates are applied to search on this field.

A sample document:

```
{
  "_id": 0,
  "ID": 1,
  "name": "April Cohen",
  "age": 22,
  "mark": 85,
  "major": "Studies in Religion",
  "description": "Background and Family Alex Johnson was born on March 14, 2001
      ..."
}
```

4.4.1.2 Workload Generation

Each workload comprises a single read query, specifically targeting retrieval operations using MongoDB's `find()` command. As this project focuses on index-based query optimisation, only read queries were examined since indexes primarily benefit retrieval operations. Designing a single query workload allows for a detailed inspection of Mindexer's internal index estimation process, enabling the examination of how Mindexer scores each candidate index against a query.

The query in the workload were designed to include diverse predicate types (equality search, range search) and sort modifiers on different fields, aiming to capture various combinations of predicates. This variety was essential for identifying scenarios where Mindexer failed to adhere to the ESR rule by varying the combinations of equality and range predicates, with or without sort modifiers, thereby effectively examining Mindexer's ability to estimate the benefits of different indexes.

The sample workload:

```
W_1 = [{"major": "Nutrition Science", "mark": {"$gt": 80, "$lt":
    95}}].sort({"name": 1})
W_2 = [{"major": "Nutrition Science", "mark": {"$gt": 80, "$lt":
    95}}].sort({"age": 1})
W_3 = [{"age": {"$gt": 18, "$lt": 30}, "major": "Computer Science"}].sort({"ID":
    1})
W_4 = [{"major": "Computer Science", "age": {"$gt": 18, "$lt": 30}}].sort({"ID":
    1})
...
```

4.4.1.3 Evaluation of Index Recommendations

For each workload, Mindexer was executed to generate its recommended index. The recommended index was first assessed to determine whether it adheres to the ESR rule. If the index does not comply with the ESR rule, both the recommended index and the index that follows the ESR rule were compared in terms of performance to determine if Mindexer is recommending a suboptimal index.

For example:

Consider the workload:

```
W_1 = [{"major": "Nutrition Science", "mark": {"$gt": 80, "$lt":
    95}}].sort({"name": 1})
```

If Mindexer recommends {'mark': 1, 'major': 1, 'name': 1}, this index fails to adhere to the ESR rule. The index that adheres to the ESR rule is {'major': 1, 'name': 1, 'mark': 1}. The performance of {'mark': 1, 'major': 1, 'name': 1} and {'major': 1, 'name': 1, 'mark': 1} was then measured and compared. If {'major': 1, 'name': 1, 'mark': 1} performs better than {'mark': 1, 'major': 1, 'name': 1}, it indicates that Mindexer is recommending a suboptimal index.

4.4.2 Experimental Execution and Control Measures

To ensure the validity of the results, several measures were implemented. Firstly, the sample rate for Mindexer was set to 1.0 to ensure optimal performance and eliminate any randomness introduced by sampling. Additionally, Mindexer was executed multiple times for each workload to verify the consistency of its output.

The comparison of index performance between Mindexer's recommendation and the index that follows the ESR rule strictly adheres to the Index Performance Assessment Methodology outlined in Section 4.1.

4.4.3 Mindexer Code Review

The source code of Mindexer was examined to elucidate the index recommending process and identify the root cause of its behaviour in recommending indexes. This code review facilitated a deeper understanding of the experimental observations and helped explain why Mindexer may recommend suboptimal indexes.

4.5 Phase 2: Proposal and Development of Index Minder's Enhanced Cost Estimation

Phase 2 aims to address the limitations identified in **Mindexer** by developing a method to differentiate between indexes with identical fields but varying orders. This involves evaluating the effectiveness of MongoDB's ESR rule through performance experiments and conducting an in-depth analysis of index structures to understand the relationship between index scan costs and the number of index entries examined. Based on these studies, a new algorithm is proposed and integrated into **Mindexer**, resulting in the enhanced tool, **Index Minder**, which provides more accurate index scan cost estimations and optimised index recommendations by accounting for field order variations.

4.5.1 Process Overview

The general process of this phase encompasses the following key steps:

(1) **Phase 1 Insights and Directions for Enhancement** Consolidating the observations and analyses from Phase 1 to determine the root causes of **Mindexer**'s limitations and outlining the directions for improvement. Specifically, it was identified that **Mindexer**'s cardinality estimator effectively captures the filter effect of indexes but employs an oversimplified index scan cost estimation. Consequently, **Mindexer** cannot differentiate between indexes containing the same fields in different orders, leading to potentially suboptimal index recommendations.

(2) **Mechanism Study on the ESR Rule and Index Structure**

 (a) **Effectiveness of the ESR Rule** Conducting experiments to evaluate whether the ESR rule is generally more effective than other field orderings (e.g., ERS, RSE) in index recommendations. For each query, indexes with different field orders were created and their performance was assessed using MongoDB's `explain` function. The key measurements recorded were the number of index entries examined and the query execution time in milliseconds. The results indicated that indexes adhering to the ESR rule tend to minimise query execution time by reducing the number of index entries examined.

 (b) **Impact of Field Order on Index Structure** Conducting an in-depth analysis of MongoDB's B-tree index structure to uncover the relationship between index scan cost and the number of index entries examined. It was determined that the index scan cost is directly proportional to the number of index entries accessed. This finding underscores the importance of field order in index design, as it significantly impacts the number of index entries that must be examined during query execution. However, the in-depth analysis revealed that the ESR rule is not always optimal. Due to concerns about recommending suboptimal indexes, the ESR rule should not be implemented as the sole basis for index recommendations.

(3) **Proposition of a New Algorithm** Designing a new cost estimation algorithm that accurately determines the number of index entries examined for a given index and query. This algorithm accounts for the field order in compound indexes and considers various query predicates. The algorithm was tested across multiple scenarios by comparing its estimates with the actual results obtained from MongoDB's `explain` function, confirming its accuracy in predicting the number of index entries examined.

(4) **Implementation of Index Minder** Integrating the proposed algorithm into **Mindexer** to develop **Index Minder** as an enhanced index recommendation tool. This integration involves:

- **Index Scan Cost Estimator**: Adding a index scan cost estimator with the new algorithm, enabling more accurate cost estimations based on the number of index entries examined.
- **Dynamic Index Scan Cost Per Entry**: An additional feature added to **Index Scan Cost Estimator** which leads to the index scan cost per index entry varies based on the number of fields in the index, further refining the accuracy of index scan cost estimations.
- **System Architecture Enhancements**: Redesigning the system architecture to accommodate the improved cost estimation model, ensuring scalability and efficiency.

- **Computational Efficiency and Memory Optimisation**: Employing optimisation techniques to ensure that **Index Minder** remains computationally efficient and memory-efficient, even when handling large workloads.

These enhancements address the weaknesses identified in **Mindexer**, particularly in distinguishing between field orders in compound indexes and providing more accurate cost estimations. Additionally, **Index Minder** introduces a feature that adjusts the index scan cost per index entry based on the number of fields in the index, further refining the accuracy of index recommendations.

4.5.2 New Algorithm for Estimate Number of Examined Index Entries

To determine the number of index entries that need to be examined for a compound index and a specific query, the following algorithm is proposed:

(1) **Initialisation**: Start with the first field in the index and proceed sequentially to the last field.

(2) **Field Analysis**: For each field, perform the following steps:

  (a) **Equality Check**: Determine if the field is used solely for an equality predicate in the query.
  - **If yes**: The field helps narrow down the number of index entries to be examined. Record the associated equality predicate and proceed to the next field.
  - **If no**: Proceed to step (b).

  (b) **Termination Condition**: Check if the field is used for a range or inequality predicate, or if it is irrelevant (not used) in the query.
  - **If the field is used for a range or inequality predicate**: It is the last field that can help narrow down the index entries. Record the associated range or inequality predicate, and terminate the checking process.
  - **If the field is irrelevant**: It cannot be used to narrow down the index entries. Terminate the checking process.

After the field checking process terminates, the number of documents satisfying all the recorded predicates represents the number of index entries that will be examined.

$$\text{Number of index entries examined} = |D(\text{recorded predicates})| \tag{4.1}$$

4.5.3 Pseudocode for Index Scan Cost Estimator

Index Scan Cost Estimator is a new estimator deployed in Index Minder, it implement the algorithm described in 4.5.2. The following pseudocode outlines the process of estimating index scan cost for each candidate index.

```
Define a constant INDEX_SCAN_COST

For each query q in workload W:
    Classify predicates into equal_predicates and non_equal_predicates
    Initialize query_for_estimate

    For each candidate index c:
        For each field in candidate index:
            if field matches a non-equality predicate:
                Add non-equality predicate to query_for_estimate and stop
            Else If field matches an equality predicate:
                Add equality predicate to query_for_estimate
            Else:
                Stop

        Estimate number of keys examined using query_for_estimate
        Calculate index scan cost:
            index_scan_cost = INDEX_SCAN_COST * number_of_fileds_in_c *
                number_of_keys_examined

        Store the index scan cost for query q and index c
```

This pseudocode describes how Index Minder evaluates the index scan cost by classifying predicates and determining the number of index keys that need to be examined for each query and candidate index. The cost per examined entry varies based the number of field, further improve the accuracy in estimating index scan cost.

4.5.4 Index Minder Index Benefit Estimate Process

Index Minder consists of three core components for index benefit estimation: the **Index Scan Cost Estimator (ISCE)**, **Cardinality Estimator (CE)**, and **Sort Estimator (SE)**. The process of estimating the benefit of a candidate index against a given query is as follows:

(1) **Index Scan Cost Estimation**: Compute the number of index entries to be examined using the **ISCE**. Multiply this result by a constant **INDEX_SCAN** (adjusted by the ISCE based on the number of fields in the index) to obtain the theoretical index scan cost.
(2) **Document Fetch Cost Estimation**: Compute the number of documents that need to be fetched using the **CE**. Multiply this result by a constant **FETCH_COST** (representing the cost of fetching each document via the index pointer) to obtain the theoretical document fetch cost.
(3) **Sort Cost Estimation**: Use the **SE** to determine whether the index can be used for sorting. If the index cannot be used for sorting, estimate the **in-memory sort cost**.
(4) **Total Cost Calculation**: Sum the theoretical costs obtained from steps 1, 2, and 3 to compute the total estimated cost for query execution using the candidate index.
(5) **Index Benefit Calculation**: Subtract the estimated **collection scan cost** from the total query execution cost to determine the benefit of the candidate index for the specific query.
(6) **Score Storage**: Store the calculated benefit in the score table for further operations to determine the recommended indexes.

The **estimated collection scan cost** is obtained by multiplying the number of documents by a constant representing the cost per document for a collection scan (**COLLECTION_SCAN**).

4.6 Phase 3: Performance Evaluation and Comparative Analysis

The final phase of this methodology involves two distinct parts. The first part is the performance evaluation of **Index Minder** itself, aiming to uncover its properties—specifically focusing on how conservative the tool is and determining the threshold of benefit required for recommending an index. The second part is a comparative performance analysis between **Index Minder** and **Mindexer**, assessing which tool provides more effective index recommendations under various workloads.

4.6.1 Part 1: Index Minder Conservativeness Evaluation

An essential aspect of evaluating an index recommender tool is understanding its *conservativeness*, which refers to the tool's threshold for recommending an index based on the expected performance benefits it can provide. A conservative index recommender requires a higher threshold of anticipated performance improvement before suggesting the creation of an index, whereas a less conservative tool may recommend indexes even for marginal gains. This evaluation aims to determine how conservative **Index Minder** is in its recommendations.

To achieve this, various scenarios will be constructed wherein indexes offer differing levels of performance enhancement. By testing whether **Index Minder** recommends indexes in these scenarios and evaluating the proportion of recommendations relative to potential benefits, it is possible to ascertain the tool's conservativeness and identify the threshold at which it deems an index sufficiently advantageous to recommend.

4.6.1.1 Experimental Design

Index performance varies significantly based on query selectivity. Therefore, by using a query with two predicates and varying the selectivity of each predicate, 121 scenarios with different index use cases can be created to demonstrate optimal index performance.

The query contains two predicates: one equality search and one range search:

```
Q = {{'major': "Computer Science"},{'mark' : {$le: <value>}}}}
```

To adjust the selectivity of each predicate, modifications were performed:

- **Range Predicate Selectivity**:
  - The placeholder `<value>` in the range predicate was replaced with values corresponding to percentiles from the 0th to the 100th percentile, in steps of 10.
  - This adjustment systematically varies the selectivity of the range predicate.
- **Equality Predicate Selectivity**:
  - Adjusting the selectivity of the equality predicate required modifying the collection data.
  - A copy of the university student collection was created.

- Starting with 0% of documents having the `'major'` field equal to `"Computer Science"`, 10% of the documents that did not originally have this value were randomly selected and updated to `"Computer Science"`.
- This process was repeated ten times, resulting in the equality predicate selectivity varying from 0% to 100% in steps of 10%.

With 11 selectivity levels for both predicates, there are 121 combinations in total, allowing the query selectivity to range from 0% to 100% with finer granularity. This comprehensive range covers the full span of query selectivity in real-world scenarios, which is useful for assessing index benefits.

Since the query consists of only equality and range predicates, and there is no correlation between the `'major'` and `'mark'` fields, the theoretical optimal index can be easily determined as `{'major': 1, 'mark': 1}`. This combination is expected to have the least number of index entries examined. The ER (Equality-Range) design of the query simplifies the process of choosing the optimal index for each scenario.

4.6.1.2 Experiment Execution

For the 121 scenarios, three measurements were recorded:

(1) The average query execution time without the index.
(2) The average query execution time with the index.
(3) Whether **Index Minder** recommended an index or not.

The execution time measurements strictly adhered to the Index Performance Assessment Methodology outlined in Section 4.1.

The execution times were used to compute the execution time reduction provided by the index:

$$\text{Time Reduction (\%)} = \frac{T_{\text{no index}} - T_{\text{with index}}}{T_{\text{no index}}} \times 100 \tag{4.2}$$

where $T_{\text{no index}}$ represents the average execution time without the index, and $T_{\text{with index}}$ represents the average execution time with the index.

4.6.1.3 Expected Experimental Results

By analysing the execution time reduction provided by the optimal index and **Index Minder**'s decisions across the 121 scenarios, the conservativeness of **Index Minder** can be determined:

(1) **Error Detection**:
- Check whether **Index Minder** makes mistakes by recommending an index when it results in negative performance effects.

(2) **Recommendation Rate**:
- In scenarios where the index brings positive effects, assess the percentage **Index Minder** suggests an index.

(3) **Benefit Threshold**:
- Determine the threshold of index benefit (time reduction) required for **Index Minder** to recommend an index.

4.6.1.4 Results Visualisation and Analysis

The results were visualised as a heatmap in Figure 5.3, which illustrates:

- **Percentage of Execution Time Reduction**:
  - Each cell in the heatmap represents a combination of equality and range selectivity.
  - The colour intensity indicates the percentage reduction in execution time achieved by the optimal index.
- **Index Minder Recommendations**:
  - Cells are annotated to indicate whether **Index Minder** recommended an index for that particular scenario.

4.6.1.5 Automated Conservativeness Adjustment

This evaluation of **Index Minder**'s conservativeness also provides a method for adjusting its conservativeness to meet specific goals. For example, if the objective is to achieve a recommendation rate of 50% when an index can bring benefit, but currently the recommendation rate is 30%, then **Index Minder** can reduce its ratio between `FETCH` and `COLLSCAN` costs to become less conservative. This adjustment process can be repeated multiple times, similar to hyperparameter tuning in machine learning, allowing **Index Minder** to eventually reach the desired level of conservativeness.

4.6.2 Part 2: Comparative Performance Analysis between Index Minder and Mindexer

The general process involves designing the collection and workloads, followed by comparing the quality of the recommended indexes from **Mindexer** and **Index Minder** under different sample rates. Additional metrics will also be evaluated.

4.6.2.1 Collection and Workload Design

Collection. The collection used in this phase is the university student synthetic dataset created in Phase 1.

Workloads. The workloads are generated using **MDBRTools** (MongoDB Labs, 2024), an experimental package for schema analysis and query workload generation utilised by MongoDB Research. This tool can control the number of predicates in a query, the fields examined in predicates, and the selectivity of predicates. However, it does not generate sort output modifiers; therefore, adjustments were made to the generated workloads to include sort output modifiers.

The principle of generating workloads was to avoid specifying selectivity explicitly, thereby mimicking real-world situations and ensuring that no inherent advantage is given to **Index Minder** (which has a medium level of conservativeness and tends to suggest indexes when the query selectivity is low).

There are two types of workloads:

- Workloads containing equality and range predicates.
- Workloads containing equality and range predicates with sort output modifiers.

Equality and Range (ER) Workload. This workload consists of queries that involve both equality and range predicates. The queries are designed to test how well the index recommender handle filtering operations without involving sorting, which is the dominant use case for indexes. The ER workload includes the following queries:

```
[
    {'mark': {'$gt': 57.0}, 'major': 'Hearing and Speech'},
    {'mark': {'$gt': 65.0}, 'name': 'Jordan Chang'},
    {'age': {'$lte': 25.0}, 'mark': {'$lte': 73.0}},
    {'name': 'Albert Ford', 'age': {'$lt': 31.0}},
    {'mark': {'$lte': 64.0}, 'name': 'Roberto Scott'},
    {'major': 'Wildlife Conservation (minor)', 'age': {'$gte': 19.0}},
    {'age': {'$lte': 13.0}, 'mark': {'$lte': 78.0}},
    {'major': 'Pathology (minor)', 'name': {'$ne': 'Jerry Graham'}},
    {'age': {'$gt': 23.0}, 'major': 'Italian Studies'},
    {'major': 'High Performance in Sport', 'name': 'Greg Benjamin'}
]
```

Equality, Sort, and Range (ESR) Workload. This workload extends the ER workload by adding a sort modifier to each query, resulting in the Equality, Sort and Range(ESR) workload. The ESR workload is designed to evaluate the performance of indexing strategies when sorting is involved in query execution.

Both **Mindexer** and **Index Minder** recommend indexes based on their expected benefit, defined as the reduction in workload execution time. When sorting is involved, the benefit of an index also includes the time reduce by eliminating the need for in-memory sorting. The time required for in-memory sorting can vary greatly due to factors such as system busyness, making the index recommendation process more challenging. Therefore, the ESR workload provides a valuable opportunity to examine how both index recommenders perform under conditions where sorting introduces additional variability. The ESR workload includes the following queries:

```
[
    {'mark': {'$gt': 57.0}, 'major': 'Hearing and Speech'}.sort([('age', 1)]),
    {'mark': {'$gt': 65.0}, 'name': 'Jordan Chang'}.sort([('major', 1)]),
    {'age': {'$lte': 25.0}, 'mark': {'$lte': 73.0}}.sort([('mark', 1)]),
    {'name': 'Albert Ford', 'age': {'$lt': 31.0}}.sort([('name', 1)]),
    {'mark': {'$lte': 64.0}, 'name': 'Roberto Scott'}.sort([('age', 1)]),
    {'major': 'Wildlife Conservation (minor)', 'age': {'$gte':
        19.0}}.sort([('major', 1)]),
    {'age': {'$lte': 13.0}, 'mark': {'$lte': 78.0}}.sort([('mark', 1)]),
    {'major': 'Pathology (minor)', 'name': {'$ne': 'Jerry Graham'}}.sort([('name',
        1)]),
    {'age': {'$gt': 23.0}, 'major': 'Italian Studies'}.sort([('age', 1)]),
    {'major': 'High Performance in Sport', 'name': 'Greg
        Benjamin'}.sort([('major', 1)])
]
```

Each query in the ESR workload includes a sort modifier, where the sort key is specified along with the sort order (1 for ascending).

4.6.2.2 Experiment Execution with Different Index Recommenders

The experiment involves executing the generated workloads with the presence of indexes suggested by **Mindexer** and **Index Minder** to compare their performance. The following scenarios were evaluated:

- **No Index**: Workloads are executed without any secondary indexes, serving as a baseline for performance comparison.
- **Indexes Recommended by Mindexer with Sample Ratio 1.0**: Workloads are executed using the indexes recommended by **Mindexer**, with the sample ratio set to 1.0. This configuration leverages the full dataset, providing the optimal performance for **Mindexer**.
- **Indexes Recommended by Index Minder with Varying Sample Ratios**: Workloads are executed using the indexes recommended by **Index Minder** at different sample ratios to observe the impact of sampling on index recommendation quality. The sample ratios used are:
  - 1.0 (using the full dataset)
  - 0.5
  - 0.2
  - 0.1
  - 0.01

By varying the sample ratios for **Index Minder**, the experiment examines how the proportion of data used for index recommendation affects the performance of the suggested indexes, thus testing the robustness of **Index Minder** under different sample ratios. This analysis is crucial for understanding the trade-offs between computational overhead in index recommendation and the performance of the suggested indexes.

4.6.3 Evaluation Metrics and Analysis of Results

The performance metrics used for evaluation include:

- **Recommended Indexes Quality**: This metric strictly adheres to the Index Performance Assessment Methodology outlined in Section 4.1.
- **Index Recommendation Time**: Assessing the running time of the index recommendation tool.
- **Index Creation Overhead**: Assessing the computational cost of creating indexes.

These metrics provide a comprehensive view of the trade-offs between improved query performance and the overhead associated with index recommendations.

The results are analysed using statistical techniques to determine the significance of performance improvements between **Mindexer** and **Index Minder**.

4.6.4 Validity and Fairness of Evaluation

To ensure the validity of the performance evaluation, the experimental design controls for variables such as hardware configurations and MongoDB versions.

4.7 Iterative Development and Future Enhancements

The methodology is designed to support iterative refinement. Based on the findings from each phase, further enhancements to Index Minder can be implemented. Future iterations may focus on expanding the tool's capabilities to handle more complex queries, incorporate write-heavy workloads, or adapt to changes in MongoDB's indexing mechanisms. In particular, extending the current model to consider new or hybrid index structures in MongoDB, such as hash or wildcard indexes, could further optimise performance.

Additionally, integrating Index Minder with MongoDB's query planner directly, or exploring machine learning-based approaches for index selection, are promising areas for future research. The iterative nature of this methodology allows the tool to evolve alongside advancements in database technology and indexing algorithms, ensuring it remains relevant and effective in a wide variety of application scenarios.

CHAPTER 5

Results

5.1 Overview

This chapter presents the findings of the research in alignment with the three phases outlined in the Methodology chapter. The results are organised as follows:

(1) **Phase 1: Performance Analysis of Mindexer**
  - **Mindexer Behaviour Observation**: An overview of how **Mindexer** operates under various workloads.
  - **Index Recommendation Quality Compared to ESR Rule**: Evaluation of the quality of indexes recommended by **Mindexer** in comparison to those adhering to the ESR rule.
  - **Source Code Review and Root Cause Analysis**: Insights from the source code examination that reveal the underlying reasons for **Mindexer**'s limitations.

(2) **Phase 2: Proposal and Development of Index Minder's Enhanced Cost Estimation**
  - **Phase 1 Insights and Directions for Enhancement**: Consolidating the observations and analyses from Phase 1 to determine the issues that need to be addressed and outlining the directions for improvement.
  - **Effectiveness of the ESR Rule through Experiments**: Experimental validation of the ESR rule's effectiveness compared to alternative field orderings.
  - **In-depth Study of Index Structure**: Analysis of the relationship between index scan costs and the number of index entries examined.
  - **Development and Testing of a New Algorithm**: Presentation and evaluation of the newly proposed algorithm for estimating the number of index entries.
  - **Implementation of Index Minder**: Description of the integration of the new algorithm into **Mindexer**, resulting in the enhanced tool, **Index Minder**.

(3) **Phase 3: Performance Evaluation and Comparative Analysis**

- **Properties of Index Minder**: Examination of the conservativeness in recommending indexes and other key properties of **Index Minder**.
- **Comparative Performance Analysis**: Comparative analysis of **Mindexer** and **Index Minder** across different sample ratios, evaluating metrics such as tool running time, index creation time, and workload execution time.

Each section within this chapter provides a detailed account of the experiments conducted, the data collected, and the analyses performed to validate the enhancements introduced in **Index Minder**. By systematically presenting the results corresponding to each phase, this chapter underscores the effectiveness of the proposed modifications and their impact on index recommendation performance in MongoDB.

5.1.1 Experimental Environment

The experiments were conducted in the following hardware and software environment:

- **Hardware**
  - Model: MacBook Pro
  - Chip: Apple M1 Pro
  - Cores: 10 (8 performance cores, 2 efficiency cores)
  - Memory: 16 GB
  - Storage:
    - Capacity: 994.66 GB
    - Available: 464.31 GB

- **Software**
  - Operating System: macOS 14.2.1
  - MongoDB Version: 7.0.2
  - Architecture: `aarch64`
  - Allocator: system

5.2 Phase 1: Performance Analysis of Mindexer

5.2.1 Mindexer Behaviour Observation

To examine **Mindexer**'s behaviour, the synthetic university student collection was utilised, and multiple workloads were executed to assess **Mindexer**'s behaviour and its index recommendations. The following four sample workloads are presented for demonstration purposes:

```
W1 = [{"major": "Nutrition Science", "mark": {"$gt": 80, "$lt":
    95}}].sort({"name": 1})
W2 = [{"major": "Nutrition Science", "mark": {"$gt": 80, "$lt": 95}}].sort({"age":
    1})
W3 = [{"age": {"$gt": 18, "$lt": 30}, "major": "Computer Science"}].sort({"ID": 1})
W4 = [{"major": "Computer Science", "age": {"$gt": 18, "$lt": 30}}].sort({"ID": 1})
...
```

The sample rate of **Mindexer** was set to 1.0 to eliminate randomness introduced by sampling. **Mindexer** was also executed multiple times to verify the robustness of its index recommendations.

5.2.1.1 Recommended Indexes

For workload W1, **Mindexer** consistently recommended the index `{'mark': 1, 'major': 1, 'name': 1}`, corresponding to the RES configuration, which does not adhere to the ESR rule. According to the ESR rule, the optimal index configuration should be `{'major': 1, 'name': 1, 'mark': 1}`.

For workload W2, **Mindexer** consistently recommended the index `{'major': 1, 'mark': 1, 'age': 1}`, corresponding to the ERS configuration, which also does not fully adhere to the ESR rule. According to the ESR rule, the optimal index configuration should be `{'major': 1, 'age': 1, 'mark': 1}`.

For workloads W3 and W4, **Mindexer** consistently recommended the index `{'major': 1, 'ID': 1, 'age': 1}`, which adheres to the ESR rule.

These observations indicate that the recommended indexes remain unchanged for a given workload when the sample rate is set to 1.0. Additionally, it was discovered that **Mindexer**'s recommendations do not consistently follow the ESR rule, resulting in various index configurations being suggested.

5.2.1.2 Recommended Indexes Quality Check

For workloads W1 and W2, where **Mindexer** recommended indexes not adhering to the ESR rule, a comparison was conducted between the **Mindexer**-recommended indexes and those adhering to the ESR rule.

Workload W1. The execution statistics for W1 with different index configurations are presented in Table 5.1.

TABLE 5.1: Workload Execution Stats for W1 with Different Index Configurations

Index Configuration	**Average Execution Time (ms)**	**Sort in Memory**
No Index	300	Yes
Mindexer's Index	125	Yes
ESR	25	No

Workload W2. The execution statistics for W2 with different index configurations are presented in Table 5.2.

TABLE 5.2: Workload Execution Stats for W2 with Different Index Configurations

Index Configuration	**Average Execution Time (ms)**	**Sort in Memory**
No Index	281	Yes
Mindexer's Index	11	Yes
ESR	7	No

For both W1 and W2, the indexes adhering to the ESR rule demonstrated superior performance, with lower execution times and in-memory sorting eliminated. Although each **Mindexer**-recommended index includes the field specified in the sort output modifier, the index does not support sorting, as the output is still sorted in memory. This provides strong evidence that **Mindexer** does not fully consider the ESR rule, leading to the recommendation of suboptimal indexes.

5.2.1.3 Source Code Review

The source code review revealed that index benefit evaluation in **Mindexer** is primarily conducted by the **cardinality estimator**. This estimator assigns higher scores to indexes that include a greater number of fields used in the query. Specifically, it only considers the presence of fields associated with the query's predicates. Consequently, as long as indexes contain the same fields, they receive identical scores from the **cardinality estimator**, preventing **Mindexer** from differentiating between them. This often results in multiple indexes sharing the highest score, with **Mindexer** randomly recommending one of them.

In general, if a query Q contains equality and range predicates such as:

$$Q = \{P_{\text{equality1}}(\text{field}_A), P_{\text{equality2}}(\text{field}_B), P_{\text{range1}}(\text{field}_C), P_{\text{range2}}(\text{field}_D)\},$$

Mindexer may suggest any index configuration that includes all the relevant fields (`{'field_A': 1, 'field_B': 1, 'field_C': 1, 'field_D': 1}`) because they all receive the highest score from the **cardinality estimator**, regardless of the optimal field order, leading to suboptimal recommendations that disregard the ESR rule.

5.3 Phase 2: Proposal and Development of Index Minder's Enhanced Cost Estimation

5.3.1 Insights from Phase 1 and Directions for Enhancement

The primary benefit of an index is its ability to reduce the number of documents that need to be examined to answer a query. **Mindexer**'s cardinality estimator accurately estimates this reduction, enabling it to suggest beneficial indexes in most cases.

However, the cardinality estimator assumes that the index scan cost is linearly related to the number of documents that need to be fetched after index scanning. This oversimplification leads to inaccurate index scan cost estimations, making it impossible for **Mindexer** to differentiate between indexes with identical fields but different field orders.

Therefore, while the cardinality estimator should be retained due to its accurate assessment of the lookup effect, modifications are necessary. Specifically, the index scan cost estimation should be decoupled from the cardinality estimator and addressed separately to enable differentiation between indexes with identical fields but varying orders.

For a comprehensive analysis, refer to Section 6.1.2.

5.3.2 Effectiveness of the ESR Rule through Experiments

There are two general use cases for the ESR rule: (1) Predicates-Only Queries and (2) Predicates-and-Sort Queries. Each use case will be examined separately to assess the effectiveness of the ESR rule. To evaluate the ESR rule's effectiveness in each scenario, specific queries and indexes with different configurations will be created. To simplify the candidate index space, each candidate index will include all fields present in the query but will differ only in field order. The synthetic university student data collection is used for these experiments.

5.3.2.1 Methodology

To assess the effectiveness of the ESR rule, the following methodology was employed:

(1) **Query and Index Configuration**: For each use case, specific queries were formulated, and multiple index configurations were created. Each index included all fields present in the query but varied in the order of these fields to test different indexing strategies.

(2) **Execution and Data Collection**: Each query was executed against the database using MongoDB's `explain()` function. The `explain()` function provides detailed execution statistics, which were extracted and analysed. Key metrics extracted include:
   - **Exec Time (ms)**: The total time taken to execute the query.
   - **Docs Fetched**: The number of documents retrieved by the query.
   - **Scan Time (ms)**: The time spent scanning the index.
   - **Entries Examined**: The number of index entries examined during the scan.
   - **Mem Sort**: Indicates whether sorting was performed in memory.

   See Subsection 10.1.2 for more details on interpreting the `explain()` output.

(3) **Data Presentation**: The collected metrics were organised into tables to facilitate comparison between different index configurations.

5.3.2.2 Predicates-Only Queries

Example Query. Consider the following query:

```
{"major": "Computer Science", "mark": {"$gt": 70}}
```

This query is expected to return 7,823 documents.

Index Configurations. Two index configurations were tested:

- **ER Index**: `{'major': 1, 'mark': 1}`
- **RE Index**: `{'mark': 1, 'major': 1}`

Execution Statistics Extraction. For each index configuration, the query was executed with the `explain()` function. The relevant execution statistics were extracted from the `executionStats` and `IXSCAN` stages of the `explain()` output.

Comparison of Query Execution Performance. The execution statistics for each index configuration are summarised in Table 5.3.

TABLE 5.3: Comparison of Query Execution Performance for Predicates-Only Queries

Index Config	Exec Time (ms)	Docs Fetched	Scan Time (ms)	Entries Examined
No Index	225	N/A	N/A	N/A
ER Index	29	7823	28	7823
RE Index	544	7823	528	894390

According to Table 5.3, the ER configuration is the optimal one among the three index configurations. The ER configuration minimises the index scan cost by reducing the number of index entries examined, resulting in significantly lower execution time compared to the RE index and no index.

5.3.2.3 Predicates-and-Sort Queries

Example Query. Consider the following query:

```
{"major": "Computer Science", "mark": {"$gt": 70}}.sort({"age":1})
```

This query is expected to return 7,823 documents sorted by age.

Index Configurations. Various index configurations were tested:

- **ESR Index**: `{'major': 1, 'age': 1, 'mark': 1}`
- **ERS Index**: `{'major': 1, 'mark': 1, 'age': 1}`
- **RSE Index**: `{'mark': 1, 'age': 1, 'major': 1}`
- **RES Index**: `{'mark': 1, 'major': 1, 'age': 1}`
- **SER Index**: `{'age': 1, 'major': 1, 'mark': 1}`
- **SRE Index**: `{'age': 1, 'mark': 1, 'major': 1}`

Execution Statistics Extraction. For each index configuration, the query was executed with the `explain()` function. The relevant execution statistics were extracted from the `executionStats`, `IXSCAN`, `FETCH`, and `SORT` stages of the `explain()` output.

Comparison of Query Execution Performance. The execution statistics for each index configuration are summarised in Table 5.4.

TABLE 5.4: Comparison of Query Execution Performance for Predicates-and-Sort Queries

Index Config	Exec Time (ms)	Docs Fetched	Scan Time (ms)	Entries Examined	Mem Sort
No Index	228	N/A	N/A	N/A	Yes
ESR Index	36	7823	34	7836	No
ERS Index	62	7823	23	7823	Yes
RSE Index	677	7823	677	894390	Yes
RES Index	669	7823	625	894390	Yes
SER Index	1276	7823	1276	1000000	No
SRE Index	1247	7823	1247	1000000	No

According to Table 5.4, the ESR configuration generally minimises the number of index entries examined while maintaining the ability to sort. It often results in the least overall query execution time. There is a positive correlation between the number of index entries examined and the index scan cost when the index sizes are similar (containing identical fields). Some index configurations negatively affect query execution time, emphasising the importance of accurate index benefit estimation to ensure the recommendation of beneficial indexes.

These observations are consistently seen with other queries involving predicates and sorting. When sorting with an index is required, the ESR rule ensures the least index scan cost. However, sometimes the increased index scan cost introduced by sorting with the index may exceed the cost of sorting in memory. In such cases, an ER index is more effective than an ESR index.

Conclusion. In summary, there is a positive correlation between the number of index entries examined and the index scan cost when the index sizes are similar (containing identical fields). The ESR rule generally minimises the index scan cost by reducing the number of index entries that need to be examined, thereby optimising query performance. However, in scenarios where the index scan cost surpasses the cost savings from sorting, alternative index configurations such as ER may provide better performance.

For a comprehensive analysis, refer to Section 6.2, which provides a detailed in-depth study of the index structure.

5.3.3 In-depth Study of Index Structure

The index structure used in MongoDB is a B-tree index. The balanced tree structure ensures a consistent search cost for every index entry (leaf node). There are two types of predicates: equality searches and range searches. Different search types result in different numbers of index entries examined or accessed. Typically, range searches cause more index entries to be examined since more entries fit the range search criteria, and range searches involve traversing multiple nodes. Theoretically, the index scan cost is positively related to the number of index entries that need to be examined for the same index.

When the index is a compound index, the index tree is built according to the order of the index configuration. A primary tree is constructed based on the first field, and at the leaf level of the primary tree, subtrees are built on the subsequent fields, and so on. Therefore, searching in a compound index tree must follow the order of the index configuration. This process can be seen as gradually narrowing down the search space, reducing the number of index entries that need to be accessed. However, if any field in the index is associated with a non-equality predicate, all nodes beneath it need to be traversed, so any following fields in the index will not contribute to narrowing down the search space.

This is the reason the ESR (Equality-Sort-Range) rule works in most cases. By placing equality fields as the prefix of an index, it minimises the number of index entries that need to be examined. However, placing the sort field between equality and range fields does enable sorting with the index but also increases the number of index entries that need to be examined, leading to a higher index scan cost. In some cases, when the number of documents required to be sorted is small, the cost of sorting in memory may be less than the additional index scan cost introduced by sorting with the index. Therefore, sorting with the index (using an ESR index) may not be the optimal solution; an ER index may be better.

Here is an example:

Query:

```
{"major": "Nutrition Science", "mark": {"$gt": 80, "$lt": 95}}.sort({"name": 1})
```

776 documents should be returned in sorted order by name.

ER index:

```
{'major': 1, 'mark': 1}
```

ESR index:

```
{'major': 1, 'name': 1, 'mark': 1}
```

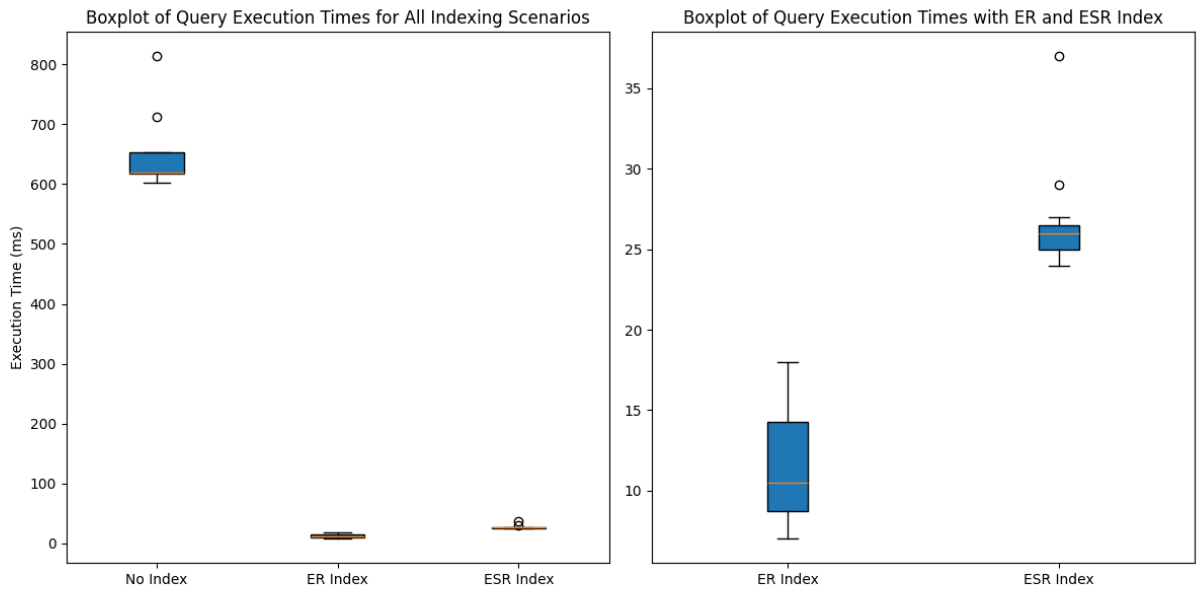

FIGURE 5.1: Boxplots of Query Execution Times for Different Indexing Scenarios

According to Figure 5.1, both ER and ESR indexes reduce query execution time, but in this case, the ER index is the optimal one, not the ESR index. This illustrates the limitation of the ESR rule and the significance of having an accurate number of index entries examined based index scan cost estimator to work alongside the sort benefit estimator to make trade-offs about whether to use the index for sorting.

For a comprehensive analysis, refer to Section 6.2, which provides a detailed in-depth study of the index structure.

5.3.4 Development and Testing of a New Algorithm

The development process and the detailed rationale behind this algorithm are provided in Section 6.2.3.3.

5.3.4.1 Algorithm for Computing the Number of Index Entries Examined

To determine the number of index entries that need to be examined for a compound index and a specific query, the following algorithm is proposed:

(1) **Initialisation**: Start with the first field in the index and proceed sequentially to the last field.
(2) **Field Analysis**: For each field, perform the following steps:
   (a) **Equality Check**: Determine if the field is used solely for an equality predicate in the query.
      - **If yes**: The field helps narrow down the number of index entries to be examined. Record the associated equality predicate and proceed to the next field.
      - **If no**: Proceed to step (b).

   (b) **Termination Condition**: Check if the field is used for a range or inequality predicate, or if it is irrelevant (not used) in the query.
      - **If the field is used for a range or inequality predicate**: It is the last field that can help narrow down the index entries. Record the associated range or inequality predicate, and terminate the checking process.
      - **If the field is irrelevant**: It cannot be used to narrow down the index entries. Terminate the checking process.

After the field checking process terminates, the number of documents satisfying all the recorded predicates represents the number of index entries that will be examined.

$$\text{Number of index entries examined} = |D(\text{recorded predicates})| \tag{5.1}$$

5.3.4.2 Testing the Algorithm

The algorithm was tested using the synthetic university student collection. Queries with multiple predicates were created, and for each query, indexes with different configurations were constructed. The number of index entries that would be examined was computed using the proposed algorithm and compared with the actual number of index entries examined, as reported by MongoDB's `explain()` function. See Subsection 10.1.2 for more details on interpreting the `explain()` output.

Sample Test Case.

Consider the query:

```
{"major": "Computer Science", "mark": {"$gt": 80, "$lt": 95}}.sort({"name": 1})
```

The following index configurations were tested:

ESR Configuration.

```
{"major": 1, "name": 1, "mark": 1}
```

According to the algorithm, the number of index entries that should be examined is equal to the number of documents satisfying the predicate `{"major": "Computer Science"}`, which is 8,757. The actual number of index entries examined, as reported by MongoDB, is 8,354.

ERS Configuration.

```
{"major": 1, "mark": 1, "name": 1}
```

According to the algorithm, the number of index entries that should be examined is equal to the number of documents satisfying the predicates
`{"major": "Computer Science", "mark": {"$gt": 80, "$lt": 95}}`, which is 941. The actual number of index entries examined is 941.

RES and RSE Configurations.

```
{"mark": 1, "major": 1, "name": 1}
{"mark": 1, "name": 1, "major": 1}
```

Applying the algorithm, the number of index entries that should be examined is equal to the number of documents satisfying the predicate `{"mark": {"$gt": 80, "$lt": 95}}`, which is 106,245. The actual number of index entries examined is 106,245.

SER and SRE Configurations.

```
{"name": 1, "major": 1, "mark": 1}
{"name": 1, "mark": 1, "major": 1}
```

According to the algorithm, both indexes require scanning the entire index. Therefore, the computed number of index entries examined is 1,000,000, and the actual number of index entries examined is 1,000,000.

This testing was conducted with multiple different queries and sets of indexes, and the same observations consistently occurred.

5.3.4.3 Results and Observations

The observations indicate that the algorithm has high accuracy in computing the number of index entries examined. In most cases, the computed values exactly match the actual values reported by MongoDB. In cases where the index configuration is ESR, the algorithm slightly overestimates, which might be due to MongoDB's performance optimisations for ESR index configurations.

These results demonstrate the effectiveness of the algorithm in predicting the number of index entries that need to be examined for various index configurations and query predicates. This accuracy is crucial for improving index recommendation tools, as it enables more precise cost estimations.

For a comprehensive analysis of the algorithm's development process, refer to Section 6.2.3.3.

5.3.5 Implementation of the Enhanced Index Recommender: Index Minder

5.3.5.1 New estimator: Index Scan Cost Estimator

The Index Scan Cost Estimator is a critical component of the Index Minder system, designed to accurately predict the index scan cost for various index configurations and query predicates. This subsection presents the pseudocode of ISCE, providing evidence that it considers both the number of index entries examined and the index size, thereby addressing the limitations of Mindexer and the ESR rule. The inclusion of index size in the estimation process is essential to capture the overhead associated with larger indexes; for a detailed discussion, please refer to Section 6.3.2.

The following pseudocode outlines the process of estimating index scan cost for each candidate index:

```
Define a constant INDEX_SCAN_COST

For each query q in workload W:
    Classify predicates into equal_predicates and non_equal_predicates
    Initialize query_for_estimate

    For each candidate index c:
        For each field in candidate index:
            if field matches a non-equality predicate:
                Add non-equality predicate to query_for_estimate and stop
            Else If field matches an equality predicate:
                Add equality predicate to query_for_estimate
            Else:
                Stop

        Estimate number of index entries examined using query_for_estimate
        Calculate index scan cost:
            index_scan_cost = INDEX_SCAN_COST * number_of_fileds_in_c *
                number_of_keys_examined

        Store the index scan cost for query q and index c
```

This pseudocode outlines how Index Minder evaluates the index scan cost. It classifies query predicates, examines index fields sequentially, and determines the number of index entries that must be scanned. The resulting number is then multiplied by a constant, which varies based on the number of index fields, to estimate the theoretical cost for each query and candidate index.

5.3.5.2 System Architecture of Index Minder

The architecture of Index Minder builds on Mindexer by introducing new components to improve the quality of index recommendations and modifying existing ones. A key enhancement is the addition of an **Index Scan Cost Estimator (ISCE)** to the benefit estimator, which now works alongside the **Cardinality Estimator (CE)** and the **Sort Estimator (SE)**. With the ISCE in place, the index scan cost estimation function is removed from the CE, as it is now managed by the ISCE. These three estimators—ISCE, CE, and SE—work together to determine the benefit of an index for a query. This enhancement addresses the limitations of Mindexer and allows Index Minder to make more informed decisions about index configurations, aiming to surpass simple adherence to the **ESR rule**. The modifications in Index Minder are highlighted in light blue in Figure 5.2.

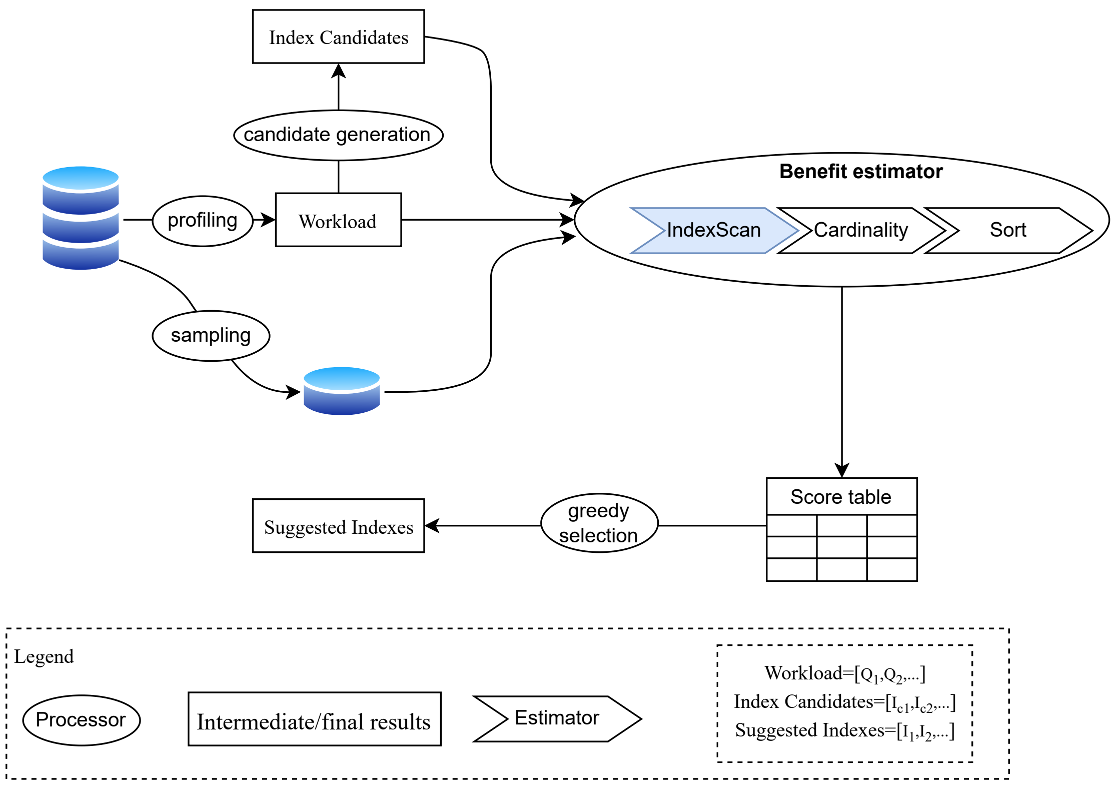

FIGURE 5.2: System Architecture of Index Minder

Compared to Mindexer (see Section 2.3.3.1), the primary improvement in Index Minder is the integration of the **Index Scan Cost Estimator (ISCE)** into the benefit estimation process. The original **Cardinality Estimator (CE)** in Mindexer could not distinguish between indexes with the same fields but in different configurations, often leading to suboptimal recommendations. With the addition of the ISCE, Index Minder works alongside the CE and the **Sort Estimator (SE)** to make more informed decisions based on the overall benefit of an index, rather than focusing on just one aspect. This collaboration allows Index Minder to wisely trade off between the added index scan cost for sorting and the cost of sorting in memory. This improvement enables Index Minder to achieve better performance than simply following the **ESR rule**.

5.3.5.3 Index Benefit Estimation with Index Minder

Index Minder consists of three core components for index benefit estimation: the **Index Scan Cost Estimator (ISCE)**, **Cardinality Estimator (CE)**, and **Sort Estimator (SE)**. The process of estimating the benefit of a candidate index against a given query is as follows:

(1) **Index Scan Cost Estimation**: Compute the number of index entries to be examined using the **ISCE**. Multiply this result by a constant **INDEX_SCAN** (adjusted by the ISCE based on the number of fields in the index) to obtain the theoretical index scan cost.

(2) **Document Fetch Cost Estimation**: Compute the number of documents that need to be fetched using the **CE**. Multiply this result by a constant **FETCH_COST** (representing the cost of fetching each document via the index pointer) to obtain the theoretical document fetch cost.

(3) **Sort Cost Estimation**: Use the **SE** to determine whether the index can be used for sorting. If the index cannot be used for sorting, estimate the **in-memory sort cost**.

(4) **Total Cost Calculation**: Sum the theoretical costs obtained from steps 1, 2, and 3 to compute the total estimated cost for query execution using the candidate index.

(5) **Index Benefit Calculation**: Subtract the estimated **collection scan cost** from the total query execution cost to determine the benefit of the candidate index for the specific query.

(6) **Score Storage**: Store the calculated benefit in the score table for further operations to determine the recommended indexes.

The **estimated collection scan cost** is obtained by multiplying the number of documents by a constant representing the cost per document for a collection scan (**COLLECTION_SCAN**).

5.3.5.4 Conservativeness Control

The constants **INDEX_SCAN**, **COLLECTION_SCAN**, and **FETCH_COST** control the conservativeness of **Index Minder**. Details on these constants are provided in Section 5.3.5.3. **INDEX_SCAN** is always smaller than **COLLECTION_SCAN** since an index is never larger than the collection. The ratio between **FETCH_COST** and **COLLECTION_SCAN** is fine-tuned for the environment to set the threshold for recommending indexes. A higher **FETCH_COST** makes Index Minder more conservative, recommending fewer indexes, while a lower value results in more recommendations. Currently, the conservativeness is set to 50%, meaning approximately only the top 50% of indexes, based on their estimated benefit, are recommended by Index Minder. For more details on conservativeness control, please refer to Section 6.3.4.

5.4 Phase 3: Performance Evaluation and Comparative Analysis

This section presents the evaluation of **Mindexer** and **Index Minder** under different query scenarios and compares their performance based on workload execution times and tool running times.

5.4.1 Properties of Index Minder

This experiment explores how Index Minder's behaviour changes with varying index performance by testing it under 121 different selectivity scenarios. These scenarios are designed to vary the performance of the theoretical best index (the ER index) across a full range of query selectivity, from 0% to 100%, with 11 discrete levels for both equality and range predicates.

For each scenario, the percentage reduction in query execution time achieved by the ER index was calculated and colour-coded on the heat map. A positive time reduction, indicating improved query performance, is represented in green, with darker shades reflecting greater improvements. A negative time reduction, where the index increases query execution time, is represented in red, with darker shades indicating greater negative impacts. The percentage time reduction can theoretically range from $-\infty$ to 1. The heat map in Figure 5.3 visualises these results.

Index Minder was also executed for each of the 121 scenarios to observe how its recommendations align with the performance of the ER index. The recommendations are indicated on the heat map (Figure 5.3) with an underscore beneath the corresponding values in cells where Index Minder suggested an index.

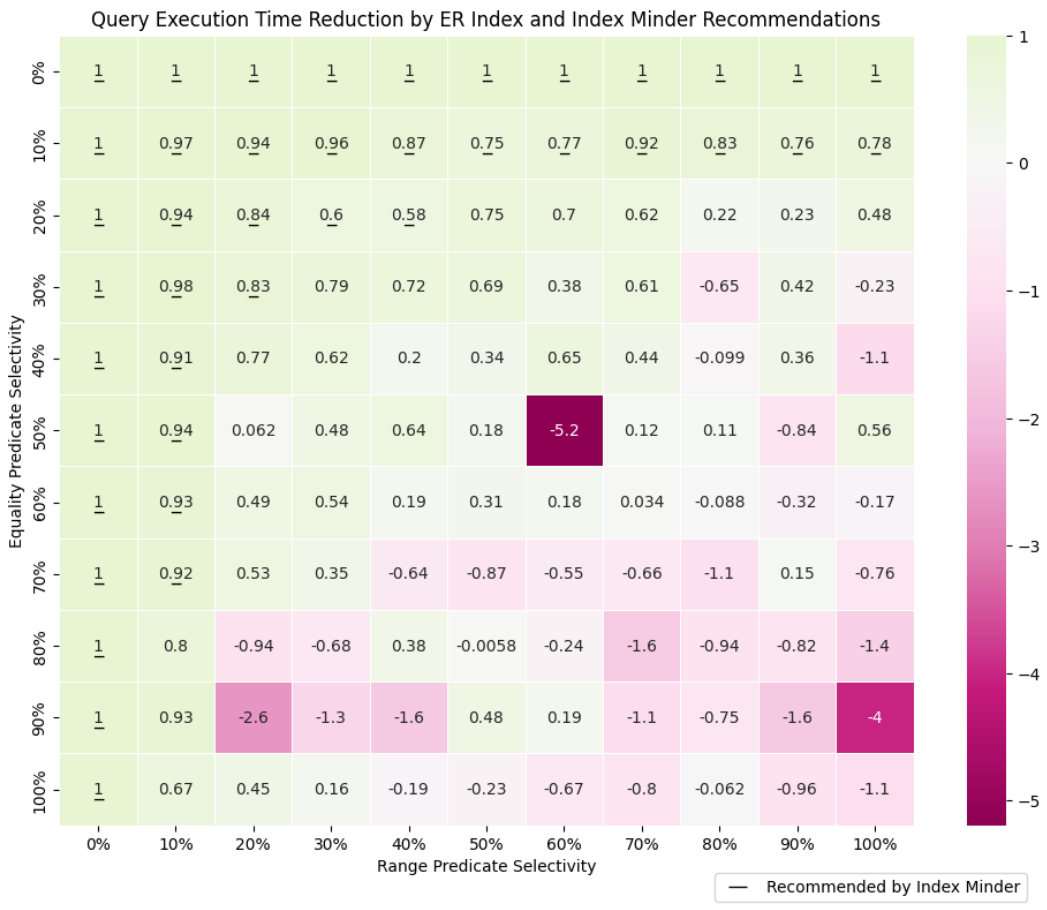

FIGURE 5.3: Query Execution Time Reduction by ER Index and Index Minder Recommendations

As shown in Figure 5.3, the ER index improves query performance in 84 out of 121 scenarios (69%), highlighting that indexes do not always result in performance gains.

In total, Index Minder recommended an index in 41 scenarios. It only provides recommendations when it determines that an index is beneficial. Of the 84 scenarios where the ER index improved query performance, Index Minder recommended an index 49% of the time (41 out of 84). The smallest time reduction for which Index Minder recommended an index was 58%.

5.4.1.1 Index Minder's Performance Beyond the ESR Rule

Recall the scenario discussed in Section 5.3.3, where the limitations of the ESR (Equality-Sort-Range) rule are evident.

Query:

```
{"major": "Nutrition Science", "mark": {"$gt": 80, "$lt": 95}}.sort({"name": 1})
```

This query returns 776 documents, sorted by `name`.

For this query, Index Minder recommends the following: **ER index**:

```
{"major": 1, "mark": 1}
```

In contrast, the ESR rule suggests: **ESR index**:

```
{"major": 1, "name": 1, "mark": 1}
```

As illustrated in Figure 5.1, the ER index recommended by Index Minder outperforms the ESR-based index. Adding a sort field (`name`) to the index in accordance with the ESR rule introduces additional index scan costs that exceed the cost of performing in-memory sorting. By accurately estimating these costs, Index Minder overcomes the limitations of the ESR rule and recommends the more efficient ER index.

5.4.2 Comparative Analysis between Mindexer and Index Minder

This section presents a comparative analysis between Mindexer and Index Minder by mainly evaluating the quality their index recommendations on synthetic datasets under different query workloads.

5.4.2.1 Collection and Workload

This experiment uses a synthetic university student collection, with workloads generated by `MDBRTools`. For details on workload generation, please refer to Section 4.6.2.1. Two types of query workloads are evaluated:

Equality and Range (ER) Workload: This workload contains queries with both equality and range predicates, testing how well indexing strategies handle filtering operations without sorting.

Equality, Range, and Sort (ESR) Workload: This workload extends the ER workload by adding a sort modifier to each query, evaluating the performance of indexing strategies under sorting operations.

5.4.2.2 Workload Execution with Different Index Strategies

The workloads were executed using various indexing strategies to compare the performance of Mindexer and Index Minder. The following indexing strategies were evaluated:

- **No Index**: Queries executed without indexes, serving as a baseline.
- **Mindexer (1.0)**: Queries executed with indexes recommended by Mindexer, using a sample ratio of 1.0 for the entire collection.
- **Index Minder (Varying Sample Ratios)**: Queries executed with indexes recommended by Index Minder, using different sample ratios to assess the impact of sampling on index recommendation quality. The sample ratios are:
  - 1.0
  - 0.5
  - 0.2
  - 0.1
  - 0.01

By varying the sample ratios, the experiment explores the trade-off between the computational cost of index recommendation and the performance of the recommended indexes. This analysis examines the robustness of Index Minder under different sampling scenarios.

The following measurements were recorded for each indexing strategy:

- **Workload Execution Time:** The time taken to execute the queries under each strategy.
- **Index Recommendation Time:** The time required for Mindexer and Index Minder to recommend indexes.
- **Indexes Recommended:** The specific indexes suggested by each tool.
- **Recommended Indexes Creation Time:** The time required to create the recommended indexes.

5.4.2.3 Equality and Range (ER) Workload

Workload Execution Time: The performance of different indexing strategies under the ER workload is summarised in Figure 5.4. This figure presents the workload execution time for each strategy, highlighting the median, interquartile range, and outliers.

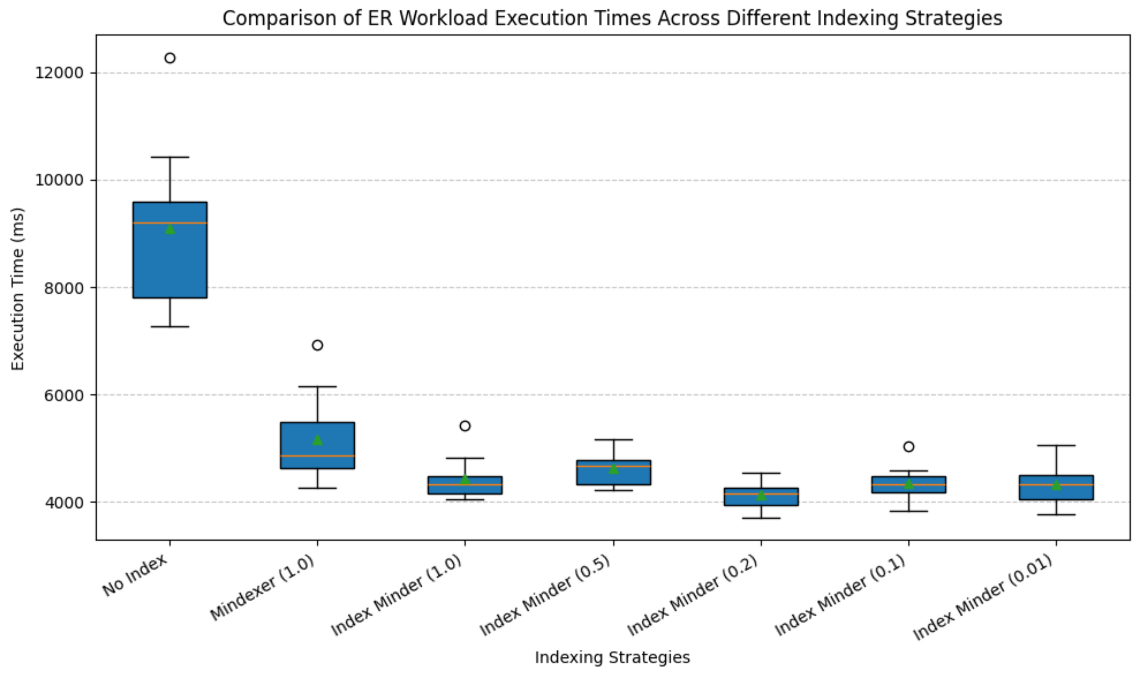

FIGURE 5.4: Performance Evaluation (ER Workload)

Statistical Test Results: Two statistical tests were performed to compare the execution times:

- **Two-sample t-test (Mindexer vs. Index Minder, Sample Ratio 1.0):**
  - t-statistic: 2.529
  - p-value: 0.025
- **ANOVA Test (Index Minder with Different Sample Ratios):**
  - F-statistic: 0.422
  - p-value: 0.738

According to Figure 5.4 and the two-sample t-test results, the workload execution time with the set of indexes recommended by Index Minder (with a sample ratio of 1.0) is significantly shorter than that with the indexes recommended by Mindexer. This indicates that the indexes recommended by Index Minder (with a sample ratio of 1.0) offer superior performance by reducing execution time more effectively than those recommended by Mindexer. Additionally, the quality of indexes recommended by Index Minder across different sample ratios does not vary significantly, suggesting that its performance is not greatly affected by changes in sample ratio.

Index Recommendation Time and Recommended Indexes Creation Time: In addition to workload execution time, the recommendation and creation times for indexes under different strategies are compared in Figure 5.5. This visualisation provides a clear comparison of the computational overhead for each strategy.

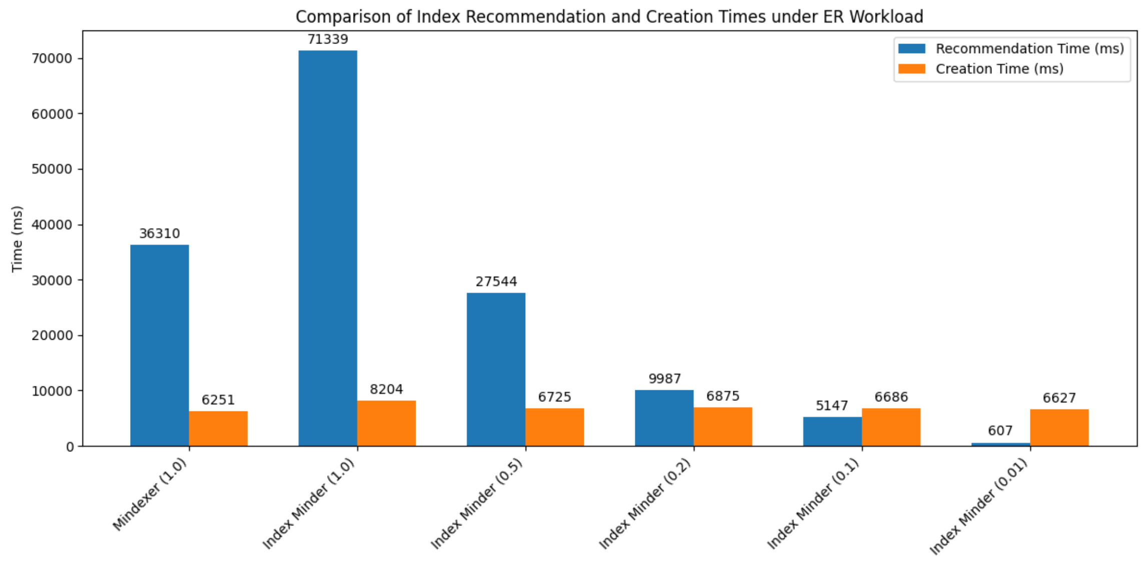

FIGURE 5.5: Comparison of Index Recommendation and Creation Times (ER Workload)

According to 5.5, the index recommendation time for Index Minder is approximately double that of Mindexer when the sample ratio is set to 1.0. The recommendation time decreases as the sample ratio decreases. The index creation time for Index Minder, across all sample ratios, is slightly longer than that of Mindexer, though not significantly. This is expected, as Index Minder consistently recommends more indexes than Mindexer.

Mindexer (Sample Ratio: 1.0).

- **Indexes Recommended:**
  - {"name": 1, "major": 1}
  - {"major": 1, "age": 1}
  - {"name": 1, "mark": 1}

Index Minder.

- **Sample Ratio: 1.0**
  - **Indexes Recommended:**
    - {"major": 1, "name": 1}
    - {"name": 1, "mark": 1}
    - {"major": 1, "age": 1}
    - {"major": 1}
    - {"name": 1}
- **Sample Ratio: 0.5**
  - **Indexes Recommended:**
    - {"major": 1, "name": 1}
    - {"name": 1}
    - {"major": 1, "age": 1}
    - {"major": 1}
- **Sample Ratio: 0.2**
  - **Indexes Recommended:**
    - {"major": 1, "name": 1}
    - {"name": 1}
    - {"major": 1, "age": 1}
    - {"major": 1}
- **Sample Ratio: 0.1**
  - **Indexes Recommended:**
    - {"major": 1, "name": 1}
    - {"name": 1, "mark": 1}
    - {"major": 1, "age": 1}
    - {"major": 1}
- **Sample Ratio: 0.01**
  - **Indexes Recommended:**
    - {"major": 1, "name": 1}
    - {"name": 1, "age": 1}
    - {"major": 1, "age": 1}
    - {"major": 1}

5.4.2.4 Equality, Range, and Sort (ESR) Workload

The performance of different indexing strategies under the ESR workload is summarised in Figure 5.6. This figure presents the workload execution time for each strategy, highlighting the median, interquartile range, and outliers.

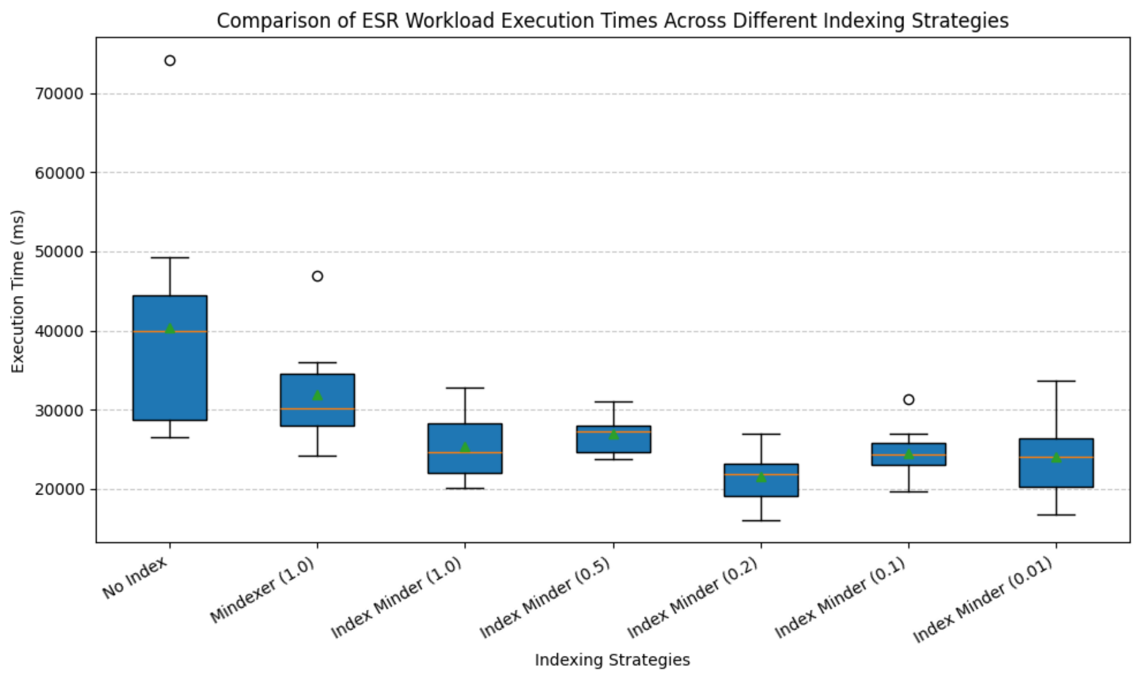

FIGURE 5.6: Performance Evaluation (ESR Workload)

Statistical Test Results: Two statistical tests were performed to compare the execution times:

- **Two-sample t-test (Mindexer vs. Index Minder, Sample Ratio 1.0):**
  - t-statistic: 2.590
  - p-value: 0.019
- **ANOVA Test (Index Minder with Different Sample Ratios):**
  - F-statistic: 2.622
  - p-value: 0.047

According to Figure 5.6 and the two-sample t-test results, the workload execution time with the set of indexes recommended by Index Minder (with a sample ratio of 1.0) is significantly shorter than that with the indexes recommended by Mindexer. This indicates that the indexes recommended by Index Minder (with a sample ratio of 1.0) offer superior performance by reducing execution time more effectively than those recommended by Mindexer. Additionally, the quality of indexes recommended by Index Minder across different sample ratios shows minimal variation, indicating that its performance remains stable across sample ratio changes. All workload execution times with indexes recommended by Index Minder, regardless of sample ratio, remain lower than those achieved with Mindexer's recommended indexes.

Index Recommendation Time and Recommended Indexes Creation Time. In addition to workload execution time, the recommendation and creation times for indexes under different strategies are compared in Figure 5.7. This visualisation offers a clear comparison of the computational overhead associated with each strategy.

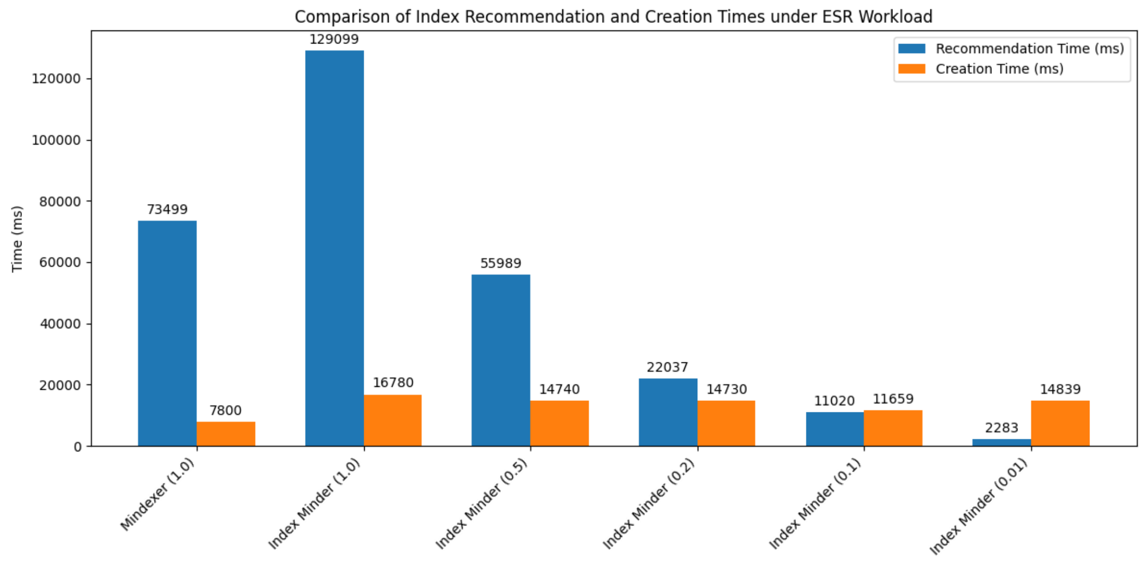

FIGURE 5.7: Comparison of Index Recommendation and Creation Times (ESR Workload)

According to 5.7, the index recommendation time for Index Minder is approximately double that of Mindexer when the sample ratio is set to 1.0. The recommendation time decreases as the sample ratio decreases. The index creation time for Index Minder, across all sample ratios, is slightly longer than that of Mindexer, though not significantly. This is expected, as Index Minder consistently recommends more indexes than Mindexer.

Mindexer (Sample Ratio: 1.0).

- **Indexes Recommended:**

  - {"mark": 1, "name": 1, "major": 1}
  - {"major": 1, "age": 1, "mark": 1}
  - {"name": 1, "major": 1, "mark": 1}
  - {"mark": 1, "age": 1}

Index Minder.

- **Sample Ratio: 1.0**
  - **Indexes Recommended:**
    - {"mark": 1, "name": 1}
    - {"major": 1, "age": 1, "mark": 1}
    - {"name": 1, "major": 1}
    - {"age": 1}
    - {"major": 1, "name": 1, "mark": 1}
    - {"mark": 1, "age": 1}
    - {"major": 1, "age": 1}
    - {"name": 1, "major": 1, "mark": 1}
    - {"name": 1, "mark": 1, "major": 1}
    - {"name": 1}
- **Sample Ratio: 0.5**
  - **Indexes Recommended:**
    - {"mark": 1, "name": 1}
    - {"major": 1, "age": 1, "mark": 1}
    - {"name": 1, "major": 1}
    - {"age": 1}
    - {"major": 1, "name": 1, "mark": 1}
    - {"mark": 1, "age": 1}
    - {"major": 1, "age": 1}
    - {"name": 1, "mark": 1}
    - {"name": 1}
- **Sample Ratio: 0.2**
  - **Indexes Recommended:**
    - {"mark": 1, "name": 1}

    * {"major": 1, "age": 1, "mark": 1}
    * {"name": 1, "major": 1}
    * {"age": 1}
    * {"major": 1, "name": 1, "mark": 1}
    * {"mark": 1, "age": 1}
    * {"major": 1, "age": 1}
    * {"name": 1, "major": 1, "mark": 1}

- **Sample Ratio: 0.1**
  - **Indexes Recommended:**
    * {"mark": 1, "name": 1}
    * {"major": 1, "age": 1, "mark": 1}
    * {"name": 1, "major": 1}
    * {"age": 1, "name": 1, "mark": 1}
    * {"major": 1, "name": 1, "mark": 1}
    * {"mark": 1, "age": 1}
    * {"major": 1, "age": 1}
- **Sample Ratio: 0.01**
  - **Indexes Recommended:**
    * {"mark": 1, "name": 1}
    * {"major": 1, "age": 1, "mark": 1}
    * {"name": 1, "major": 1}
    * {"age": 1, "name": 1}
    * {"major": 1, "name": 1, "mark": 1}
    * {"mark": 1, "age": 1}
    * {"name": 1, "mark": 1, "major": 1}
    * {"major": 1, "age": 1}

CHAPTER 6

Discussion

This chapter presents a comprehensive discussion of the findings from each phase of the research, providing essential theoretical support for the design and methodological decisions that shaped the development of the Index Scan Cost Estimator. Structured to align with the three phases of this study, the chapter begins with the *Essence of Indexing in MongoDB*, where the experimental results observed in Phase 1 are analysed to uncover the root causes and identify areas for improvement, setting the direction for Phase 2.

The following section, *The Nature of B-Tree Indexes and the ESR Rule*, delivers the in-depth theoretical insights obtained from Phase 2's mechanism study. This analysis of B-Tree structures and evaluation of the ESR Rule informs the rationale behind the new Index Scan Cost Estimator, justifying specific design choices while clarifying why certain alternatives were not adopted. The *Overcoming Limitations with Index Minder* section then addresses the strategic decisions involved in implementing the tool, detailing how the new estimator was integrated to address Mindexer's limitations.

Finally, the *Performance Analysis* section interprets the results from Phase 3, offering a comparative analysis of Index Minder and Mindexer. This section provides a nuanced examination of Index Minder's properties, discussing its strengths, limitations, and future potential as revealed through the comprehensive performance evaluation.

6.1 Essence of Indexing in MongoDB

6.1.1 Primary Role of Indexes

The primary purpose of using indexes is to **eliminate the need for a collection scan**. Without indexes, a query must perform a complete scan of the collection, which means every document in the targeted collection has to be loaded into primary storage (memory) from secondary storage (e.g., hard disks or

SSDs). The query's predicates are then evaluated against each document, and any document that meets all the specified conditions will be returned. This process is often computationally expensive and time-consuming, especially as collections can be too large to fit entirely in memory, making frequent data loading from secondary storage necessary—a costly operation.

Indexes serve as **efficient lookup guides** that allow queries to locate relevant documents without scanning the entire collection, significantly reducing the number of documents that must be examined by using the indexed fields to determine which documents need to be fetched. The figure below illustrates this concept by showing the general process of using an index to filter down the documents that need to be examined.

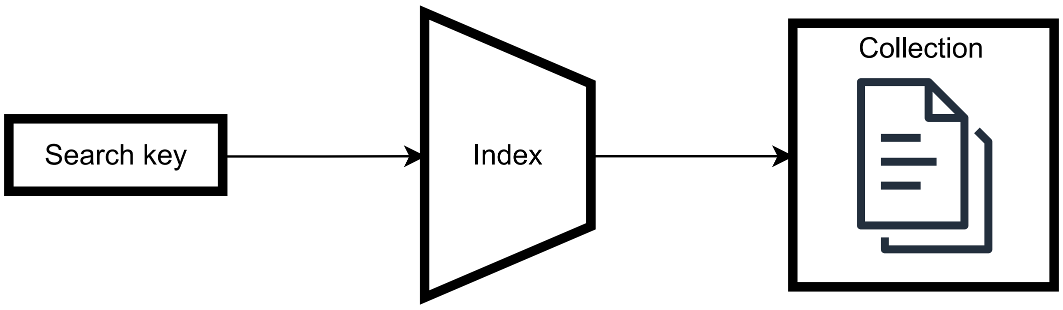

FIGURE 6.1: Illustration of index usage in query optimisation.

In the presence of an index, the **condition checking** is performed against the index, rather than directly on the entire collection. This allows the query to determine the desired documents efficiently, as the index provides pointers to the relevant documents, meaning only those matching documents need to be loaded into memory. Because indexes are designed to be small enough to reside in memory, the access speed is significantly faster, making it a cost-effective operation.

It is important to note that **not all fields in the predicates need to be included in the index**—having at least one field is sufficient for the index to improve query performance. However, the effectiveness of an index in reducing the number of documents that need to be examined depends on the **number of query fields covered by the index**.

To better understand this, let us consider a scenario where a query contains multiple predicates: $P_1, P_2, P_3, \ldots$. The documents that meet each predicate form subsets of the collection:

- $D(P_1)$: Documents that meet predicate P_1
- $D(P_2)$: Documents that meet predicate P_2
- $D(P_3)$: Documents that meet predicate P_3

The **set of documents that match all predicates** can be represented as the **intersection** of the sets corresponding to each predicate:

$$D(Q) = D(P_1) \cap D(P_2) \cap D(P_3) \cap \dots \tag{6.1}$$

Where D_Q is the set of documents that satisfy all predicates of the query Q.

The **cardinality** of the resulting set of documents (i.e., the number of documents that meet all the conditions) is denoted as:

$$|D(Q)| = |D(P_1) \cap D(P_2) \cap D(P_3) \cap \dots| \tag{6.2}$$

We observe that as more predicates are applied, the number of matching documents generally **decreases**:

$$|D(P_1)| \geq |D(P_1) \cap D(P_2)| \geq |D(P_1) \cap D(P_2) \cap D(P_3)| \geq \dots \tag{6.3}$$

The effectiveness of an index in reducing the number of documents to be examined depends on how well the index matches the predicates in the query. Specifically, the **number of documents that need to be scanned/fetched** (denoted by $|D_{\text{index}}|$) when using an index is determined by the **union of all predicates that share common fields with the index**:

$$|D_{\text{index}}| = |D(P_{\text{common}})| \tag{6.4}$$

Where P_{common} represents the set of predicates that have fields in common with the given index.

The fewer documents that need to be scanned, the greater the **cost savings** from avoiding a collection scan. Therefore, an index that contains more fields from the query predicates will generally be able to **filter down** the documents more effectively, thus minimizing the number of documents that need to be loaded and examined.

6.1.2 Link to Mindexer

Mindexer effectively estimates the **lookup effect** of a given index through its **cardinality estimator**. The cardinality estimator can compute $|D_{\text{index}}|$ (as mentioned in Equation 6.4) accurately by running queries on a sampled collection. This value, $|D_{\text{index}}|$, is then multiplied by a constant that represents the index scan cost plus the fetch cost for each document. The product provides an estimate of the total cost for the index scan as well as fetching the targeted documents identified by the index.

Since Mindexer captures the dominant benefit of an index—the reduction in documents that need to be examined—it has the potential to recommend beneficial indexes. However, the **index scan cost** is overly simplified. Mindexer assumes that this cost has a linear relationship with the number of documents determined by the index, which is not always accurate (this will be discussed in the next section). In some cases, the **index scan cost may exceed the cost saved by its lookup effect**, leading Mindexer to recommend an index that ultimately has a negative impact on performance. Due to this incorrect estimation of the index scan cost, Mindexer treats indexes with the same fields but different orders as having identical costs, since $|D(P_{\text{common}})|$ remains the same. This is the primary reason Mindexer does not adhere to the **ESR rule**—it is **insensitive to field order** within indexes. As a result, Mindexer estimates that indexes containing the same fields have the same use cost/benefit, making them indistinguishable from each other.

Despite this flaw, the **cardinality estimator** will be retained in Index Minder, as it accurately captures the lookup effect. The constant used for cost estimation will be modified to improve accuracy. Additionally, the **sort benefit** calculation will also be kept, as it correctly estimates the cost savings an index can provide by enabling sorting (sorting with indexes will be discussed in the next section).

6.2 The Nature of B-Tree Indexes and the ESR Rule

6.2.1 B-Tree Overview

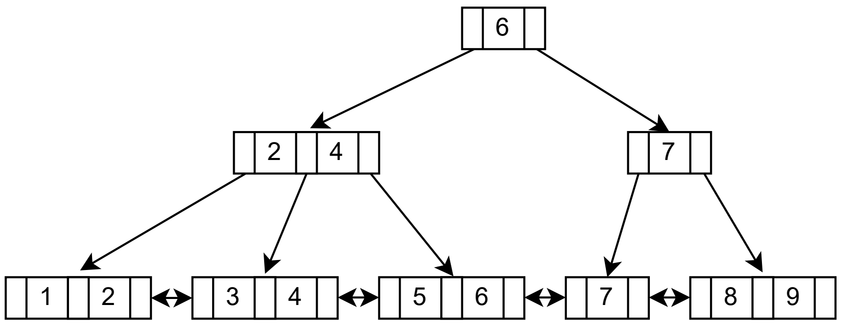

FIGURE 6.2: Illustration of B-Tree Index Structure

A **B-tree index** is the primary index structure used in MongoDB. B-tree is a **balanced search tree**, meaning it maintains an equal depth from the root node to each leaf node (which contains index entries or keys). This structure ensures that the **top-down search** cost for each leaf node is consistent. A **top-down search** refers to the process of navigating from the root of the tree to a leaf node for a specific search value, traversing each level of the tree.

The B-tree structure inherently keeps all **leaf nodes sorted** by the index fields. Moreover, each leaf node has a **double link** to its adjacent nodes, allowing for efficient traversal to neighboring nodes at a relatively low cost compared to performing a full search from the root to a leaf node. This linked structure, combined with the tree's inherent sorting of leaf nodes, enables efficient range traversals.

Due to the structure of the B-tree, **equality searches** can be implemented very efficiently if the equality search field is a **prefix** of the index. The search starts with a **top-down search** from the root node, proceeding downwards through each level of the tree. If a valid **leaf node** is found, the system uses pointers to fetch the corresponding documents. If no valid leaf node is found, the system concludes that no documents match the query predicates.

Range searches involve performing a **top-down search** to locate a **boundary leaf node**, which can be either the lower or upper boundary of the range. Once the boundary node is found, the system begins **traversing** from the boundary leaf node in the appropriate direction. The traversal continues until a leaf node is encountered that falls outside of the valid range, at which point the traversal stops. The **sorted property of the B-tree** ensures that all valid leaf nodes can be found during this traversal. Once all valid leaf nodes have been identified, the corresponding documents can be fetched using pointers.

There is a commonality between using indexes for equality searches and range searches: only a single **top-down search** is required to locate the target node or boundary, ensuring efficient retrieval in both scenarios.

6.2.2 Compound Index

A **compound index** in MongoDB consists of multiple fields, enabling queries to use a single index to efficiently filter based on multiple predicates. The way a compound index is constructed can be visualised as a **hierarchical B-tree structure** that organises the index entries field by field.

In a compound index, the first field forms the primary **B-tree**, where the index entries are initially organised based on the values of this field. At the **leaf level** of this primary B-tree, **sub-trees** are constructed for each value, using the second field in the compound index. These sub-trees further organise the index keys by the values of the second field. This process continues for each subsequent field in the compound index, effectively forming multiple levels of organisation.

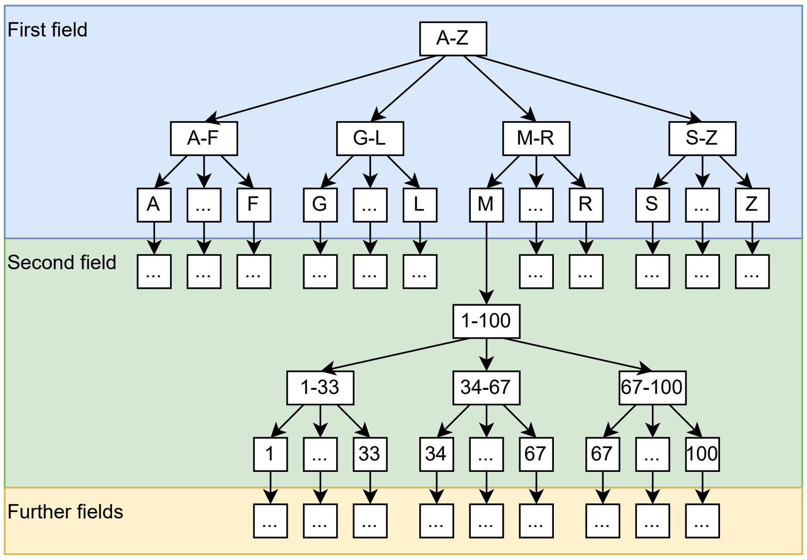

FIGURE 6.3: Illustration of the Structure of a Compound B-Tree Index

For example, consider a compound index defined as {field1: 1, field2: 1, field3: 1}. The B-tree is constructed as follows (see Figure 6.3):

- **Primary B-tree**: The first B-tree is built using **field1**, which sorts all the index entries based on their **field1** values.
- **Sub-trees at the leaf level**: At the leaf level of the primary B-tree, a **sub-tree** is built using **field2** for each unique value of **field1**. This sub-tree organises the index entries further by **field2**.
- **Further sub-trees**: If there are additional fields (e.g., **field3**), sub-trees are built at the leaf level of the previous sub-tree, organising the index keys based on **field3**.

This hierarchical B-tree structure ensures that the index entries are sorted in a specific order: first by **field1**, then by **field2** (in the case of ties in **field1**), and so on. This construction is what allows the compound index to be highly effective in filtering documents for queries with multiple predicates, as it narrows down the search space progressively at each level.

6.2.3 Compound Index Usage and Cost

Due to the **nested B-tree structure** of compound indexes, value checking must follow the order in which the index is configured, starting with the **primary B-tree** and progressing through **sub-trees** in the sequence defined by the compound index. This continues until reaching the **leaf level** of the deepest sub-tree.

When a query contains only **equality search predicates** and there is an index that contains all of the fields used in these predicates, the index search follows a **top-down search** from the root of the primary B-tree, progressing through each subsequent sub-tree until it reaches the leaf level. In this case, every field in the compound index contributes to the index search, helping to efficiently narrow down to a unique combination of values. This results in a single top-down search, leveraging all fields in the index to minimise the search space.

However, if the compound index contains **irrelevant fields**—fields that are not included in the query predicates—the index scan process changes significantly. Since there are no restrictions on these irrelevant fields, the system will treat any value for such a field as valid. This means that when reaching the sub-tree corresponding to an irrelevant field, the entire range of values under that sub-tree must be traversed. Any subsequent sub-tree built upon the irrelevant field sub-tree will not further contribute to narrowing down the search, as all index entries at this level must be traversed regardless.

A similar situation arises with **range search fields**. When the system encounters a sub-tree corresponding to a field used in a range predicate, it will locate a **boundary leaf node** within the sub-tree (either the lower or upper boundary) and begin **traversing** from that point. Because a range condition involves multiple possible values, every valid index entry within the range will be traversed. As a result, any sub-tree constructed beneath this range field will also not contribute to the index search, as all possible values will need to be traversed.

6.2.3.1 Example: Impact of Field Order on Leaf Node Traversal

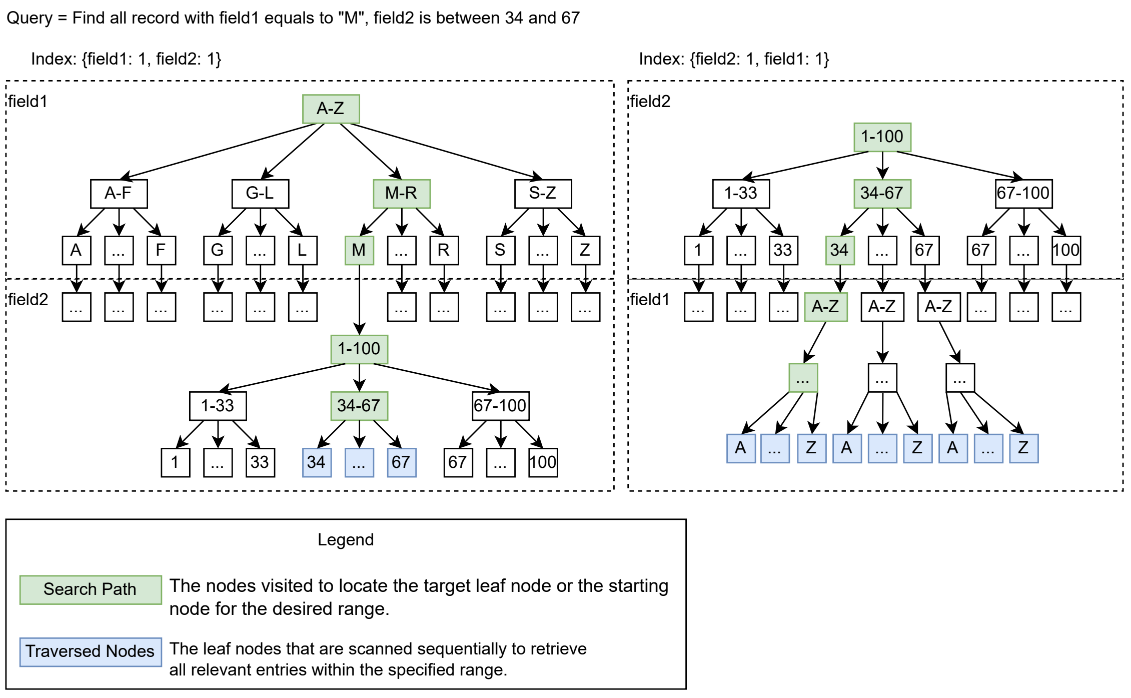
Query = Find all record with field1 equals to "M", field2 is between 34 and 67
Index: {field1: 1, field2: 1}
Index: {field2: 1, field1: 1}
field1
field2
A-Z
A-F
G-L
M-R
S-Z
1-100
1-33
34-67
67-100
Legend
Search Path
The nodes visited to locate the target leaf node or the starting node for the desired range.
Traversed Nodes
The leaf nodes that are scanned sequentially to retrieve all relevant entries within the specified range.

FIGURE 6.4: Effect of Field Order on Index Search and Traverse Paths in B-Tree Structures

To illustrate the effect of field order on the number of leaf nodes visited, consider a query with the criteria: *Find all records where field1 equals "M" and field2 is between 34 and 67*. Figure 6.4 demonstrates this query's execution with two different index configurations: {field1: 1, field2: 1} and {field2: 1, field1: 1}. When using the {field1: 1, field2: 1} index, fewer leaf nodes and index entries are traversed compared to {field2: 1, field1: 1}. This occurs because the equality predicate on field1 significantly narrows the search space right from the top of the B-tree, allowing the range predicate on field2 to operate within a smaller subset of entries. In contrast, with the {field2: 1, field1: 1} index, the system begins with the range predicate on field2, resulting in a broader traversal of leaf nodes to satisfy the query.

6.2.3.2 Summary of the Effects of Field Types on Index Usage

- **Equality Search Fields**: These fields allow the system to **navigate directly to a specific leaf node**, which represents a unique value for that field. If there are further sub-trees at the leaf node, the system continues the search within those sub-trees.
- **Range Search Fields**: These fields direct the system to locate a **boundary leaf node** and then traverse all valid index entries below it within the specified range.
- **Irrelevant Fields**: When a field is irrelevant to the query, the system must **traverse all index entries** under the corresponding sub-tree, as there are no conditions restricting these values. This makes any deeper sub-tree ineffective in further narrowing the search.

Inequality search filed has the same effect as range search field.

6.2.3.3 Cost Analysis

The commonality across all scenarios is that there is always one top-down search either results finding the final unique matching node or the node to start traversing with. This means that the initial index scan cost is the same when the fields of the indexes are identical. What differs in each scenario is how many index entries are accessed or examined. The higher the number of index entries examined, the higher the index scan cost.

To determine the number of index entries that need to be examined for a compound index and a specific query, the following algorithm is proposed:

(1) **Initialisation**: Start with the first field in the index and proceed sequentially to the last field.
(2) **Field Analysis**: For each field, perform the following steps:
   (a) **Equality Check**: Determine if the field is used solely for an equality predicate in the query.
      - **If yes**: The field helps narrow down the number of index entries to be examined. Record the associated equality predicate and proceed to the next field.

- **If no**: Proceed to step (b).

(b) **Termination Condition**: Check if the field is used for a range or inequality predicate, or if it is irrelevant (not used) in the query.

- **If the field is used for a range or inequality predicate**: It is the last field that can help narrow down the index entries. Record the associated range or inequality predicate, and terminate the checking process.
- **If the field is irrelevant**: It cannot be used to narrow down the index entries. Terminate the checking process.

After the field checking process terminates, the number of documents satisfying all the recorded predicates represents the number of index entries that will be examined.

$$\text{Number of index entries examined} = |D(\text{reordered predicates})| \tag{6.5}$$

Looking at the determination process, for a given query, if traversal can occur as late as possible, more equality fields can be used to reduce the number of index entries that need to be examined. This is the rationale behind the **ESR** rule, where **Equality (E)** fields are placed before **Range (R)** fields, and Sorting (S) will be discussed in the next subsection.

Consider a query $Q = \{P_{\text{equality1}}(\text{field}_1), P_{\text{equality2}}(\text{field}_2), P_{\text{range1}}(\text{field}_3)\}$. The **ER** configuration generally aims to minimise the number of index entries examined, as expressed in the following relationship:

$$|D(P_{\text{equality1}}, P_{\text{equality2}}, P_{\text{range1}})| \leq |D(P_{\text{equality1}}, P_{\text{range1}})| \leq |D(P_{\text{range1}})|$$

This inequality indicates that adding more equality predicates typically reduces the number of index entries that need to be examined. By ordering all the **Equality (E)** fields before the **Range (R)** field, the index will have the potential to narrow down the search space effectively.

However, the **ER** rule is not always optimal. In certain scenarios, the equality predicates may not significantly reduce the search space compared to using the range predicate alone. For instance, if:

$$|D(P_{\text{equality1}}, P_{\text{equality2}}, P_{\text{range1}})| = |D(P_{\text{range1}})|$$

then the indexes $\{\text{field}_1 : 1, \text{field}_2 : 1, \text{field}_3 : 1\}$ and $\{\text{field}_3 : 1\}$ will examine the same number of index entries. In such a case, the larger index that adheres to the **ER** rule does not provide an efficiency gain, and the index scan cost is higher due to the increased index size. When the number of index entries examined is the same, more fields result in a higher scan cost.

In practice, multiple indexes may require examining the same number of index entries. To further differentiate these indexes, the index size—which is determined by the number of fields the index contains—should also be considered to achieve more accurate index scan cost estimations.

This demonstrates that the **ESR** rule is not always perfect, as it does not always suggest the index with the lowest index scan cost. Moreover, when there are multiple range fields, the **ESR** rule does not provide guidance on how to arrange these fields effectively. Therefore, having an **accurate index scan cost estimator** is crucial for determining the efficiency of the index and selecting the optimal configuration.

6.2.4 Sorting with Indexes

6.2.4.1 Conditions for Index Sorting

In order for an index to be used for sorting, **only equality fields** should be placed before the **sort fields** in the compound index. This is due to the nested B-tree structure of the index. The equality fields narrow down the search space to a single leaf node. Starting from this leaf node, all subsequent index entries are sorted based on the next field in the index configuration. If the next field is a sort field, it will initiate a **traverse**, meaning that all relevant index entries will be examined in the order defined by the sort field.

However, if there is any **non-equality field** (either a range field or an irrelevant field) preceding the sort field (S) in the index, the traverse initiated by that non-equality field results in accessing index entries that are not sorted by S. Consequently, S does not contribute to sorting, and **the index scan cost increases** because more index entries need to be examined. As a result, sorting must be performed in memory, which **adds additional cost to the query execution**.

6.2.4.2 Impact of Sorting on Index Scan Cost

When a sort field is not used in an equality or range search, it will be considered as an **irrelevant field** from index scan cost point of view. Adding this irrelevant field to the index configuration increases the **index scan cost** because it increases the number of index entries examined.

6.2.4.3 Trade-Off Between Index Sorting and In-Memory Sorting

Sorting with an index is not always advantageous. If the number of **returned documents** is relatively small, performing the sort in memory may be more efficient, as the **increased index scan cost** exceeds the **cost of sorting in memory**. There is also an example in the results chapter 5.3.3 showing that an **ER index** can be more efficient than an **ESR index** when the number of returned documents is about 800. This illustrates that the ESR rule is not always optimal, emphasising the need for a mechanism to determine the trade-off between the increased index scan cost of sorting with the index and the cost of sorting in memory. Accurately determining this trade-off is crucial to ensure that the most efficient approach is used for the specific workload.

6.3 Overcoming Limitations with Index Minder

6.3.1 Flaw of Mindexer and the ESR Rule

The study began by experimenting with **Mindexer** to evaluate its adherence to the **ESR** rule, which is recommended by **MongoDB** for index creation. It was observed that **Mindexer** does not consistently follow the **ESR** rule due to its oversimplified index scan cost estimation, which is unable to differentiate between indexes with the same fields but in different orders.

Further studies were conducted on the **index scan cost** and the **ESR** rule. As discussed in Section 6.2, while the **ESR** rule is recommended in MongoDB's documentation for database administrators to create indexes manually, it has the potential to be helpful in most use cases. However, it is not a golden rule for optimal indexing. In many scenarios, the **ESR** rule may fail to suggest indexes with the lowest theoretical usage cost. In certain cases, blindly adhering to the **ESR** rule may even result in a negative impact on performance. Therefore, adding a feature to **Mindexer** that enforces adherence to the **ESR** rule would be a risky and unsound approach, rather than a genuine improvement.

6.3.2 Index Scan Cost Estimator

Query execution using indexes generally involves three types of costs: **index scan cost**, **document fetch cost**, and optionally **in-memory sort cost**. Accurately computing the theoretical cost is crucial for estimating the benefit of an index. **Document fetch cost** and **in-memory sort cost** can be handled by the improved **cardinality estimator** and **sort estimator** from **Mindexer**, which **Index Minder** retains with improvements. The **index scan cost**, however, is managed by the newly developed **Index Scan Cost Estimator (ISCE)**.

The ISCE computes the theoretical number of index entries that need to be examined for a candidate query and a specific workload using the process described in Section 6.2 (see Equation 6.5). However, determining only the number of index entries examined is insufficient for accurately differentiating between indexes.

Consider a given query $Q = \{P_{\text{equality1}}(\text{field}_1), P_{\text{equality2}}(\text{field}_2), P_{\text{range1}}(\text{field}_3)\}$. For two candidate indexes $\{\text{field}_1 : 1, \text{field}_3 : 1, \text{field}_2 : 1\}$ and $\{\text{field}_1 : 1, \text{field}_3 : 1\}$, both indexes require the same number of index entries to be examined, as the field after the first range field cannot be used to further narrow down the search space. Both indexes require $|D(P_{\text{equality1}}, P_{\text{range1}})|$ entries to be examined.

To accurately differentiate between such indexes, the **index scan cost** is adjusted based on the number of fields in the index. The more fields present, the larger the index size, and consequently, the higher the cost to access each index entry. With this adjustment, the ISCE is capable of accurately computing the theoretical **index scan cost** for each candidate index against each query in the workload.

6.3.3 Index Benefit Estimation

Index Minder consists of three core components for index benefit estimation: the **Index Scan Cost Estimator (ISCE)**, **Cardinality Estimator (CE)**, and **Sort Estimator (SE)**. A detailed explanation of the process of estimating the benefit of a candidate index against a given query is provided in Section 5.3.5.3 of the results chapter.

By adopting this theoretical cost estimation approach, the limitations of Mindexer and rule-based approaches (such as simply following the **ESR** rule) are avoided.

6.3.4 Controlling Conservativeness in Index Recommendations

The constants **INDEX_SCAN**, **COLLECTION_SCAN**, and **FETCH_COST** are used to control the conservativeness of **Index Minder**. Detailed explanations of these constants can be found in Section 5.3.5.3 of the results chapter. The constant **INDEX_SCAN** is always set to be smaller than **COLLECTION_SCAN**, as the size of an index will never exceed that of the collection. The ratio between

FETCH_COST and **COLLECTION_SCAN** is fine-tuned for the development environment to control the **conservativeness** of **Index Minder**—i.e., the threshold for recommending an index based on the estimated benefit it provides. A higher **FETCH_COST** value results in a more **conservative Index Minder**, recommending fewer indexes, while a lower value increases the number of recommended indexes.

There are three primary reasons for carefully controlling the conservativeness of **Index Minder**:

(1) **System Complexity and Query Execution Variance**:
Due to the complexity of computer systems, query execution times can fluctuate significantly, leading to large variances in performance. In cases where an index provides only a small theoretical benefit, this benefit may be overshadowed by system variability, resulting in the index having no effect or even a negative impact on performance. Controlling conservativeness helps mitigate this risk by ensuring that only indexes with substantial benefits are recommended.

(2) **Balancing Index Benefits and Maintenance Overhead**:
Controlling conservativeness also helps manage the trade-off between the benefits an index provides for read queries and the maintenance overhead it incurs for update workloads (i.e., Create, Update, and Delete operations). If the system handles more update queries, the conservativeness level should be set to **high**—meaning that only indexes with significant benefits are recommended, thereby reducing the number of indexes to minimise maintenance overhead. Conversely, if the system primarily handles read-heavy workloads, the conservativeness can be set to **low**, allowing more indexes to be recommended to improve read query performance.
Currently, **Index Minder** makes these decisions in a **static way**, meaning that the level of conservativeness is fixed and consistent across all given workloads. However, there is potential to make this decision-making process **dynamic**, allowing **Index Minder** to adapt based on the specific workload characteristics (e.g., read-heavy vs. update-heavy workloads). This dynamic approach will be discussed in more detail in future work.

(3) **Adaptability Across Different Environments**:
The actual collection scan cost and document fetch cost may vary from machine to machine, depending on factors such as hardware configuration and system workload. By providing a mechanism to control conservativeness, **Index Minder** can potentially adapt to different environments by automatically adjusting its conservativeness level. This adaptability could allow **Index Minder** to accurately reflect the true relationship between index scan cost and

collection scan cost across different systems, which will be explored in more detail in future work.

6.4 Performance Analysis

6.4.1 Index Minder Conservativeness: A Heat Map Analysis

6.4.1.1 Index Time Reduction Analysis

The heat map presented in Figure 5.3 illustrates the **query execution time reduction** achieved using an **Equality-Range (ER) index**. The heat map contains 11 levels of **selectivity** for each of the two predicates—**Equality** and **Range**—ranging from **0% to 100%** with a step size of **10%**, resulting in a total of **121 combinations**.

The **index time reduction percentage** is represented for each combination of selectivity levels. There is a notable **unclear boundary** that runs approximately along the **diagonal** from the lower left to the upper right of the heat map. Most of the cells **above the diagonal** have positive percentages, indicating that the **ER index** improved query execution efficiency in these cases. In contrast, most of the cells **below the diagonal** have negative values, indicating that the **ER index** had a negative impact on execution time.

The existence of the **unclear boundary** demonstrates the large variance in query execution times, which can be attributed to the inherent **complexity of computer systems**. When the index benefit is marginal, these variances may outweigh the theoretical benefit, potentially causing the index to become ineffective or even detrimental.

6.4.1.2 Conservativeness of Index Minder

The heat map also helps reveal the **conservativeness** of **Index Minder**. Cells marked with an underscore indicate cases where **Index Minder** recommended the **ER index**. Based on the visual representation, it is evident that **Index Minder** exhibits a **medium level of conservativeness**, recommending indexes only **49%** of the time when the **ER index** could potentially bring a performance benefit.

Notably, the indexes recommended by **Index Minder** are consistently **far from the boundary**, indicating that they are **stable** and have a high degree of confidence in their effectiveness, thus avoiding unpredictable impacts due to system variability. This level of conservativeness is intended to mitigate the risk of recommending indexes that are close to the **unclear boundary**, where effectiveness is uncertain. Additionally, it takes into consideration the **overhead associated with index maintenance** in update-heavy workloads, ensuring a balanced approach to recommending indexes.

Future work aims to further optimise this conservativeness level, potentially making it more adaptive to different workloads and system configurations. By doing so, **Index Minder** can better balance the trade-off between the benefit of reducing query execution time and the risk of introducing excessive overhead.

6.4.2 Comparative Performance Analysis

In both ER and ESR workload performance test scenarios, Index Minder demonstrates significantly superior performance compared to Mindexer, indicating that the improvements made in Index Minder are effective. Moreover, the performance of Index Minder is robust to variations in sample ratio, meaning it can operate with a low sample ratio, such as 0.01, to reduce the index recommendation time to a fraction of the original while still producing high-quality indexes comparable to those recommended by Mindexer with a sample ratio of 1.0.

6.4.2.1 Robustness in Performance with Sample Ratio

The robust performance of Index Minder across different sample ratios can be attributed to its index scan cost estimation process. A critical factor influencing the estimated index scan cost for a candidate index is the type of predicates in the query, as this directly affects the number of index entries that need to be examined. Since predicate types remain unchanged regardless of the sample ratio, the index search pattern also remains consistent. Specifically, only non-equality predicates trigger traversal through the index, leading to more entries being examined.

Additionally, the modified cardinality estimator and sort estimator are also designed to be robust against the effects of random sampling, further contributing to the stability of Index Minder's performance.

6.4.2.2 Index Minder Tends to Recommend More Indexes

The results also show that Index Minder tends to recommend more indexes compared to Mindexer. Mindexer cannot distinguish between indexes with identical fields in different orders, resulting in several indexes being grouped into buckets with the same estimated benefit. From each bucket, only one index is randomly selected for recommendation, which limits the number of recommended indexes and may lead to suboptimal choices, potentially even selecting indexes with a negative impact on performance.

In contrast, Index Minder's enhanced benefit estimation process provides more accurate evaluations, allowing it to capture subtle differences between indexes. This precision makes the indexes more distinct from a benefit perspective, resulting in the recommendation of more potentially beneficial indexes. The conservative control mechanism in Index Minder ensures that only indexes with meaningful performance gains are recommended, preventing the inclusion of those with marginal benefits.

6.4.2.3 Use of Indexes Leads to More Reliable Query Execution

Across both the ER and ESR workloads, the results indicate that using indexes not only reduces the execution time but also decreases variance in performance. This improvement occurs because indexes are often pre-loaded into memory, reducing the need for disk access during query execution. Consequently, index scans minimise data transfer between disk and memory, simplifying query processing and enhancing stability.

When sorting is involved, using an index to eliminate in-memory sorting further reduces variance. Sorting in memory introduces additional complexity, increasing the variability of query execution time. By leveraging indexes to handle sorting operations, the query execution becomes more predictable and stable.

CHAPTER 7

Threats to Validity

This chapter identifies potential threats to the validity of this research. These threats are categorised into internal, external, and conclusion validity. For each category, the discussion focuses on how the research design addresses these challenges to minimise their impact and ensure the reliability of the study's findings.

7.1 Internal Validity

Internal validity concerns whether the research design and execution accurately capture what the study aims to investigate.

7.1.1 Experimental Design

Throughout the study, the performance of indexes is assessed and compared, with performance defined by the reduction in workload execution time. Therefore, using workload execution time as a performance metric is a valid and direct method for evaluating index efficiency. However, due to the complexity of computer systems, execution times can fluctuate, resulting in variance across measurements. To mitigate this, each scenario was measured 11 times to capture reliable performance data, and the runtime environment was carefully controlled to maintain similar conditions. All comparative measurements were taken within short time intervals, with system status closely monitored to ensure consistency.

Additionally, statistical tests were conducted to ensure robust comparisons. The workloads used in the evaluation were generated randomly without intentional bias towards any specific predicates that might favour Index Minder. This impartial approach strengthens the validity of the results, ensuring that the findings accurately reflect the tool's performance.

7.2 External Validity

External validity refers to the extent to which the findings of this study can be generalised to other contexts, such as different datasets, workloads, or environments.

7.2.1 Synthetic Dataset

This study utilised a synthetic dataset where each field followed a controlled distribution. While this approach ensures a clean and consistent dataset for evaluation, real-world datasets often contain skewed data, outliers, or inconsistencies, which could affect Index Minder's performance. As a result, the findings may not directly apply to scenarios involving noisy or heavily skewed data.

7.2.2 Conservativeness Level

The conservativeness level of Index Minder was fine-tuned for the development environment used in this study. However, the optimal ratio between **FETCH_COST** and **COLLECTION_SCAN** may vary across different hardware configurations or workloads. As such, the conservativeness level observed on the development machine may not accurately reflect performance on other systems, potentially impacting the applicability of the results in different environments.

7.2.3 Read-Only Workload

The comparative performance analysis focused exclusively on read-only workloads. While indexes improve query performance, they also introduce maintenance overhead for update operations (such as insertions, updates, and deletions). Therefore, the results presented in this study may not fully generalise to scenarios where workloads contain a significant proportion of update queries. Future work could include evaluations on mixed workloads to provide a more comprehensive view of Index Minder's performance.

7.3 Conclusion Validity

The conclusion validity of this research refers to the degree to which the study's conclusions are credible and well-supported.

Despite the limitations discussed in the external validity section, the methodology used in this study ensures reliable and credible findings within the intended scope—evaluating the performance of index recommendation tools on read-only workloads using synthetic data. The rigorous experimental design, including multiple measurements and statistical tests, builds confidence in the evaluation results.

While the study's conclusions are valid within the defined scope, further experimentation across different datasets, workloads, and environments would enhance the generalisability of the findings. This future work would provide additional insights into the broader applicability of Index Minder's recommendations in various real-world scenarios.

CHAPTER 8

Limitations & Future Work

8.1 Limitations

This section outlines the primary limitations of the index recommendation tool developed in this thesis, highlighting areas where improvements or adjustments may be necessary.

8.1.1 Dependency on B-tree Indexes

The biggest limitation of the tool is its reliance on MongoDB's **B-tree** index structure. This dependency was introduced in Index Minder, which optimises index recommendations by focusing on **index scan cost estimation** for B-tree indexes. In contrast, Mindexer used only cardinality estimation to determine how many documents needed to be fetched after using an index, making it more generalisable to different index types. If MongoDB were to introduce alternative index structures, such as hash or inverted indexes, the current tool would require significant modifications to accommodate these changes. The tool's applicability is thus limited to environments that continue to use B-tree indexes.

8.1.2 MongoDB's Index Selection

Another important limitation lies in MongoDB's internal query optimisation strategies. The MongoDB **query planner** does not always use the recommended index, even if it is optimised for a specific query. This is due to the way MongoDB's query planner evaluates execution plans—by running them during a trial period and selecting the plan that returns the most documents in the least amount of time. In some cases, the query planner may choose an index that is suboptimal, or even harmful, for answering queries. Based on real-world experience, MongoDB sometimes selects indexes that degrade query performance, highlighting a potential conflict between the tool's recommendations and MongoDB's actual query execution behaviour.

8.1.3 Read-Only Focus

The current version of the tool is primarily designed for **read** queries (i.e., analytical workloads) and does not account for **update**, **insert**, or **delete** operations (C, U, D from CRUD). In workloads with frequent updates or writes, the presence of indexes can slow down these operations due to the overhead of maintaining the indexes. As such, while Index Minder performs well with workloads dominated by read queries, it may not provide significant performance benefits for workloads that involve frequent updates. The tool's focus on read-heavy, analytical workloads limits its applicability in environments with mixed or write-intensive workloads.

8.1.4 Complex Queries

Handling queries with many predicates introduces additional complexity. As the number of predicates increases, the number of potential index candidates grows exponentially, making it computationally expensive to evaluate all combinations. This leads to increased computation time, particularly for workloads involving highly complex queries. While the tool remains effective for simpler queries, its performance may degrade as query complexity increases, limiting its scalability in more demanding scenarios.

8.2 Future Work

The limitations identified in the previous section provide a number of opportunities for future work to enhance the tool's performance and applicability. This section outlines several potential improvements corresponding to each limitation.

8.2.1 Supporting Different Index Structures

One key area for future work is to extend the tool's support beyond **B-tree** indexes. Implementing **index scan cost estimation** for different index structures, such as **hash** or **inverted indexes**, would significantly improve the tool's flexibility. Additionally, Index Minder could be adapted for use in other NoSQL databases that rely on B-tree indexes, such as Neo4j (Neo4j, Inc., 2024). This would enable the tool to function in a wider range of database environments, increasing its applicability.

8.2.2 Ensuring MongoDB Utilises the Recommended Index

Given that MongoDB's **query planner** may not always select the recommended index, one possible improvement is to specify the target query for each recommended index. This would allow users to leverage MongoDB's `hint()` method to ensure that the query uses the recommended index. Furthermore, a deeper study of MongoDB's query planner mechanism could help in designing indexes that are not only optimal in terms of performance but also appear as the best performing option to the query planner itself.

8.2.3 Penalty Estimation for CUD Operations

To address the tool's **read-only focus**, future work should focus on constructing an accurate and efficient **penalty estimation function** for **update**, **insert**, and **delete** (C, U, D) queries. By estimating the overhead associated with maintaining indexes during these operations, the tool could provide more comprehensive recommendations for mixed workloads that involve frequent write operations.

8.2.4 Addressing Read-Only Focus: Penalty Estimation for CUD Operations

To expand beyond its current **read-only focus**, future work should aim to incorporate the impact of **update**, **insert**, and **delete** (CUD) operations on index performance. One approach involves constructing an accurate and efficient **penalty estimation function** to quantify the additional overhead of maintaining indexes during these operations. By estimating the cost of index maintenance for each type of CUD query, Index Minder could provide more comprehensive recommendations suited to mixed workloads that include frequent write operations.

Alternatively, a simpler approach would be to dynamically adjust the **conservativeness level** of index recommendations based on the workload composition. In this approach, the more update queries present in the workload, the more conservative Index Minder becomes in recommending indexes. This method saves computational time by bypassing specific cost estimations for CUD operations; however, it may not achieve the same level of accuracy as a direct cost-based penalty estimation. Despite this limitation, the dynamic conservativeness approach offers a feasible balance between computational efficiency and adaptability to different workload types.

8.2.5 Handling Complex Queries with the ESR Rule

To handle the challenge of **complex queries** with many predicates, future work could focus on recommending indexes following the **ESR rule**. While this approach may not always yield the optimal index, it would significantly reduce the time required for index estimation. This trade-off between index optimality and computational efficiency would be particularly beneficial for workloads where time is a critical factor.

CHAPTER 9

Conclusion

This research successfully developed an enhanced index recommender, Index Minder, demonstrating that a more accurate overall index usage cost estimation can significantly improve the quality of recommended indexes and, consequently, query performance. A comprehensive index usage cost estimation should account for multiple factors: index scan cost, the lookup effect, and the benefits of sorting with an index. The research introduced a novel and precise method for estimating index scan cost, which considers both the number of index entries examined and the size of the index. This method is not limited to MongoDB; it has the potential to be adapted to other database systems that use B-tree structures for indexing, highlighting the broader applicability of the approach.

The study also found that MongoDB's officially recommended ESR rule is not always optimal. The cost-based index benefit estimation process employed by Index Minder consistently outperforms simple rule-based recommendations, providing more accurate and effective index configurations.

Despite its achievements, Index Minder has certain limitations. Due to its dependency on B-tree index structures, the tool is not currently applicable to databases using alternative index structure. Furthermore, the tool focuses on read-only workloads, with the impact of indexes on update operations (e.g., insertions, updates, and deletions) managed indirectly through a static conservativeness control. This rigid control does not adapt dynamically to different workloads or environments.

Future research should aim to address these limitations. A key area for improvement is the development of an accurate and efficient penalty estimation function for update, insert, and delete (C, U, D) queries. By accurately estimating the maintenance overhead associated with indexes during write operations, the tool could offer more comprehensive recommendations for mixed workloads. Additionally, the current static conservativeness level could be enhanced by introducing a dynamic adjustment mechanism. This could involve a hyperparameter-tuning-like process that enables Index Minder to adapt to different hardware and software environments, further improving its applicability and effectiveness across diverse settings.

In summary, this research makes a significant contribution to the field by advancing the state of index recommendation through Index Minder. The tool's innovative cost estimation method and improved index benefit evaluation process address critical limitations of existing solutions, positioning Index Minder as a powerful tool for optimising query performance by indexing. With further enhancements, Index Minder has the potential to become a versatile, adaptive solution for a wide range of database systems and workloads.

CHAPTER 10

Appendix

10.1 Interpretation of MongoDB `explain()` Output

This appendix provides detailed explanations for specific metrics extracted from MongoDB's `explain()` function, as highlighted in the main text. Understanding these metrics is crucial for interpreting the performance implications of different index configurations.

10.1.1 Key Metrics Explained

Total Query Execution Time (executionTimeMillis): This metric represents the total time, in milliseconds, taken to execute the entire query. It encompasses all stages of query processing, including index scanning, document fetching, and sorting. A lower execution time indicates a more efficient query execution.

Index Scan Time (executionTimeMillisEstimate): This value estimates the time, in milliseconds, spent scanning the index to identify relevant index entries that satisfy the query predicates. It reflects the efficiency of the index in retrieving the necessary data. Accurate estimation of index scan time is essential for evaluating the performance benefits of different index configurations.

Number of Index Entries Examined (keysExamined): This metric indicates the total number of index entries that were examined during the index scan stage. It provides insight into the extent of data traversal required to satisfy the query. A lower number of entries examined typically correlates with faster query execution, as fewer data points need to be processed.

Number of Documents Fetched (nReturned): This value represents the number of documents retrieved by the query from the database. It is particularly relevant in the `FETCH` stage, where documents matching the index scan are fetched to be returned to the client. Ensuring that the number of documents fetched aligns with expectations is important for validating query correctness and performance.

10.1.2 Example `explain()` Output Interpretation

To illustrate how these metrics are extracted and interpreted, consider the following example query and its corresponding `explain()` output:

```
{"major": "Computer Science", "mark": {"$gt": 70}}.sort({"age":1})
```

ER Index: `{'major': 1, 'mark': 1}`

10.1.2.1 Overall Execution Stats

```
"executionStats": {
    "executionSuccess": true,
    "nReturned": 7823,
    "executionTimeMillis": 58, # Total query execution time
    "totalKeysExamined": 7823,
    "totalDocsExamined": 7823,
    ...
}
```

Total Query Execution Time (executionTimeMillis): The query took 58 milliseconds to execute from start to finish.

Number of Documents Fetched (nReturned): The query returned 7,823 documents.

10.1.2.2 Index Scan Stats

```
{
 "stage": "IXSCAN",
 "planNodeId": 1,
 "nReturned": 7823,
 "executionTimeMillisEstimate": 52,  # Index Scan Time
 ...
 "docsExamined": 0,
 "keysExamined": 7824  # Number of Index Entries Examined
}
```

Index Scan Time (executionTimeMillisEstimate): The index scan took an estimated 52 milliseconds to complete.

Number of Index Entries Examined (keysExamined): A total of 7,824 index entries were examined during the index scan.

10.1.2.3 Fetch Stats

```
{
 "stage": "FETCH",
 "planNodeId": 2,
 "nReturned": 7823,  # Number of Documents Fetched
 "executionTimeMillisEstimate": 1,
 ...
}
```

Number of Documents Fetched (nReturned): All 7,823 documents matching the query were fetched from the database.

Execution Time (executionTimeMillisEstimate): The fetch stage took an estimated 1 millisecond.

10.1.2.4 Sort Stats (If Applicable)

```
{
 "stage": "SORT",
 "planNodeId": 3,
 "nReturned": 7823,
 "executionTimeMillisEstimate": 26,
 ...
}
```

Execution Time (executionTimeMillisEstimate): The sort operation took an estimated 26 milliseconds to complete in memory.

10.1.3 Summary of Interpretation

In the provided example, the total query execution time is 58 milliseconds. The majority of this time (52 ms) is attributed to the index scan, where 7,824 index entries were examined. The fetch stage, responsible for retrieving the documents, took only 1 millisecond. If sorting is required and performed in memory, it adds an additional 26 milliseconds.

These metrics collectively indicate the efficiency of the index configuration. The ER index effectively minimizes the number of index entries examined, leading to a lower index scan time and overall faster query execution compared to alternative index configurations that may require examining a larger number of entries.

Bibliography

Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 1185–1196. Association for Computing Machinery, New York, NY, USA.

BSON. 2024. BSON Specification. `https://bsonspec.org`. Accessed: October 6, 2024.

S. Chaudhuri, M. Datar, and V. Narasayya. 2004. Index selection for databases: a hardness study and a principled heuristic solution. *IEEE Transactions on Knowledge and Data Engineering*, 16(11):1313–1323.

Surajit Chaudhuri and Vivek R. Narasayya. 1997. An efficient cost-driven index selection tool for microsoft sql server. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, page 146–155. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Sudarshan S. Chawathe. 2020. Estimating predicate selectivities in a nosql database service. In *2020 11th IEEE Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, pages 0414–0420.

S. Choenni, H. Blanken, and T. Chang. 1993. Index selection in relational databases. In *Proceedings of ICCI'93: 5th International Conference on Computing and Information*, pages 491–496.

E. F. Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387.

DB-Engines. 2024. DB-Engines Ranking. `https://db-engines.com/en/ranking`. Accessed: October 6, 2024.

Dipankar Deb, Rupam Dey, and Valentina E. Balas. 2019. *Engineering Research Methodology: A Practical Insight for Researchers*. Springer Singapore, first edition.

Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. Ai meets ai: Leveraging query executions to improve index recommendations. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1241–1258. Association for Computing Machinery, New York, NY, USA.

H. Gupta, V. Harinarayan, A. Rajaraman, and J.D. Ullman. 1997. Index selection for olap. In *Proceedings 13th International Conference on Data Engineering*, pages 208–219.

Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? an experimental evaluation of index selection algorithms. *Proc. VLDB Endow.*, 13(12):2382–2395.

Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215.

MongoDB Documentation. 2024. Equality, Sort, Range, and Rule. https://www.mongodb.com/docs/manual/tutorial/equality-sort-range-rule/. Accessed: April 27, 2024.

MongoDB Documentation. No date. Suggest Indexes in MongoDB. https://www.mongodb.com/docs/ops-manager/current/tutorial/suggest-indexes/. Accessed: April 27, 2024.

MongoDB, Inc. 2024a. CRUD Operations - MongoDB Manual. https://www.mongodb.com/docs/manual/crud/. Accessed: October 6, 2024.

MongoDB, Inc. 2024b. Manage the Database Profiler - MongoDB Manual. https://www.mongodb.com/docs/manual/tutorial/manage-the-database-profiler/. Accessed: October 6, 2024.

MongoDB, Inc. 2024c. MongoDB. https://www.mongodb.com. Accessed: April 27, 2024.

MongoDB, Inc. 2024d. MongoDB Atlas Performance Advisor - Index Suggestions. https://www.mongodb.com/docs/atlas/performance-advisor/#index-suggestions. Accessed: October 6, 2024.

MongoDB, Inc. 2024e. Query Plans - MongoDB Manual. https://www.mongodb.com/docs/manual/core/query-plans/. Accessed: October 6, 2024.

MongoDB, Inc. No date. MongoDB Basics: Types of Databases. https://www.mongodb.com/resources/basics/databases/types#:~:text=MongoDB%20is%20a%20general%2Dpurpose,built%20for%20modern%20application%20developers. Accessed: April 27, 2024.

MongoDB Labs. 2024. mdbrtools - MongoDB Labs. https://github.com/mongodb-labs/mdbrtools. Tool for generating workloads automatically to a MongoDB collection. Accessed: October 6, 2024.

Raghunath Nambiar and Meikel Poess. 2013. Keeping the tpc relevant! *Proc. VLDB Endow.*, 6(11):1186–1187.

Neo4j, Inc. 2024. Index Limitations and Workarounds - Neo4j Knowledge Base. https://neo4j.com/developer/kb/index-limitations-and-workaround/. Accessed: October 6, 2024.

Stratos Papadomanolakis, Debabrata Dash, and Anastasia Ailamaki. 2007. Efficient use of the query optimizer for automated physical design. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, page 1093–1104. VLDB Endowment.

Zachary Parker, Scott Poe, and Susan V. Vrbsky. 2013. Comparing nosql mongodb to an sql db. In *Proceedings of the 51st ACM Southeast Conference*, ACMSE '13. Association for Computing Machinery, New York, NY, USA.

Mohiuddin Abdul Qader, Shiwen Cheng, and Vagelis Hristidis. 2018. A comparative study of secondary indexing techniques in lsm-based nosql databases. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 551–566. Association for Computing Machinery, New York, NY, USA.

Thomas Rueckstiess. 2023. mindexer: A fast, scalable text indexer for MongoDB. `https://github.com/mongodb-labs/mindexer`. Accessed: April 27, 2024.

Karl Schnaitter, Neoklis Polyzotis, and Lise Getoor. 2009. Index interactions in physical design tuning: modeling, analysis, and applications. *Proc. VLDB Endow.*, 2(1):1234–1245.

Jiachen Shi, Gao Cong, and Xiao-Li Li. 2022. Learned index benefits: Machine learning based index performance estimation. *Proc. VLDB Endow.*, 15(13):3950–3962.

Avinash Thirukumaran. 2023. *Evaluating an Index Recommender on a Real Workload*. Bachelor of engineering (honours) thesis, The University of Sydney, Australia. Supervisors: Prof. Alan Fekete, Dr. Yash Shrivastava, Dr. Michael Cahill, Dr. Thomas Rueckstiess.