# The University of Sydney
# Faculty of Engineering
# School of Electrical and Information Engineering

## Year

A thesis submitted in partial fulfilment of a Degree

| | |
|---|---|
| **Student Name** | |
| **SID** | |
| **Unit of Study Code and Name** | |
| **Supervisor** | |
| **Title** | |

© Copyright 2020

The University of Sydney

SCHOOL OF ELECTRICAL AND INFORMATION ENGINEERING

# PROJECT CLEARANCE FORM

**Unit of Study Code and Name:**
**ELEC4713 – Thesis B**

**This is to certify that my student**

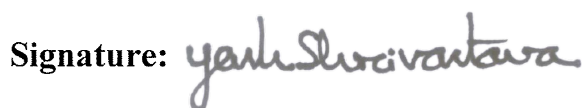
**Student Name: Avinash Thirukumaran**

**SID: 500474262**

**Has:**

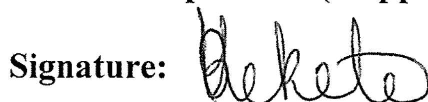
- **Returned all books and reference material;**
- **Returned all equipment and keys;** **and**
- **Tidied their work place.**

**SUEIE Academic supervisor:**

**Signature:**  **Date:** October 31, 2023

**Name: Yash Shrivastava**

**External supervisor (if applicable):**

**Signature:**  **Date:** 27/10/2023

**Name: Alan Fekete**

# Statement of Achievements

My work on this thesis resulted in two major contributions in relation to the Mindexer index recommender, which was the main subject of this study. Mindexer v0.2.0 was evaluated on a system with MongoDB as the database and with LinkBench as the real workload benchmark.

The first contribution defines the approach used to evaluate the Mindexer recommender, which I achieved after various tests and fixes, by comparing the performance of what is recommended to what is obtained by a known complex index that is beneficial (the covering index). A few scripts were written to automate this process, and this approach was utilised multiple times throughout the study to obtain results.

The second major contribution identifies several weaknesses in the design of Mindexer's scoring system and the implementation of its recommendation algorithm. This is divided into three sub-contributions. The first of these is the discovery of a bug in the implementation of its recommendation algorithm where it recommended known negatively performing indexes. This bug, alongside two bugs affecting the generation of candidate indexes and the provision of a scoring bonus, was fixed in an update to the codebase with Mindexer v0.3.0. The second sub-contribution is the identification of a flaw in the design of Mindexer's scoring system that shows an imprecision in its cardinality estimates. The third sub-contribution is the identification of a weakness in the design of Mindexer's scoring system that resulted in it variably recommending indexes of differing performance, allowing multiple unequal indexes to share the same final score. I identify various future works that could address these last two sub-contributions at the end of this thesis, which includes both defined improvements that should bring immediate improvements, as well as potential research that has less predictable outcomes.

The fixes included in the Mindexer v0.3.0 update were worked on in collaboration with Thomas Rueckstiess, who is the original developer behind Mindexer, which is itself still an experimental tool.

# Evaluating an Index Recommender on a Real Workload

Avinash Thirukumaran

A thesis presented for the degree of
Bachelor of Engineering (Honours) (Software)

School of Electrical and Computer Engineering
Faculty of Engineering
The University of Sydney
Australia
3 November 2023

Supervisor: Prof. Alan Fekete

Supervisor: Dr. Yash Shrivastava (Nominal)

Co-Supervisor: Dr. Michael Cahill

Co-Supervisor: Dr. Thomas Rueckstiess

# Declaration

## Statement of Originality

This is to certify that to the best of my knowledge, the content of this thesis is my own work. This thesis has not been submitted for any other purposes.

I certify that the intellectual content of this thesis is the product of my own work and that all assistance received in preparing this thesis and the sources used have been acknowledged.

**Name**: Avinash Thirukumaran

**Signature**:

**Date**: 3 November 2023

# Abstract

Selecting efficient indexes that can meaningfully improve the performance of a database workload is an important problem to solve, and the experimental Mindexer tool aims to solve it while utilising a small sample of the database. In this study, we evaluate the effectiveness of its index recommendations when tested on the LinkBench workload, which has a developer-recommended covering index. We present a well-tested approach to evaluating the recommender, by comparing the performance of its recommendations with that of the covering index. We identify several weaknesses in the implementation and design of its systems, consisting of an issue with negatively performing index recommendations, an imprecision in its cardinality estimates, and a flaw that variably recommends indexes of differing performance. The findings demonstrate that Mindexer does not reliably recommend viable indexes on the LinkBench workload, and we conclude that further work is needed in order for it to accurately produce meaningful results.

# Acknowledgements

Firstly, I would like to thank Alan Fekete, my supervisor, for his knowledge and guidance throughout my time working on this thesis. His support and feedback have been incredibly important leading up to this submission, and I would not have been able to deliver this work at this level of quality without him.

I would also like to thank Michael Cahill for proposing this project and guiding my initial work on this thesis. His support and knowledge of MongoDB resulted in a smooth start as I learned how to carry out the work required for this thesis.

A massive thank you goes to Thomas Rueckstiess, the creator of Mindexer from MongoDB, for all the help he has provided to me in understanding and working with the Mindexer codebase. I truly appreciate his continuous feedback and suggestions for improvements towards my work on this thesis.

Thank you to Yash Shrivastava for acting as my nominal supervisor, overseeing my progress, and providing support for any issues I encountered while writing this thesis. I would also like to thank the University of Sydney for providing this honours opportunity.

Finally, I would like to thank my family and friends for supporting me throughout the year as I have worked on this thesis. This would not have been possible without their motivation and love.

# Contents

**List of Figures** **vii**

**List of Tables** **viii**

**1 Introduction** **1**

**2 Background** **3**

**3 Literature Review** **6**
3.1 Database Performance . . . 6
3.2 OLAP Index Selection . . . 7
3.3 Automated RDBMS Index Selection . . . 8
3.4 Relational Database Index Selection . . . 8
3.5 Cost-Driven Index Selection . . . 9
3.6 The Hardness of Index Selection . . . 10
3.7 Evaluating Index Selection Algorithms . . . 11
3.8 Index Selection using Reinforcement Learning . . . 12
3.9 Scalable Index Tuning . . . 13

**4 Methodology** **15**
4.1 Contribution: Approach to Evaluating the Recommender . . . 15
4.2 Further Insights . . . 17

**5 Results** **21**
5.1 Contribution: Identification of the Weaknesses in the Design of the Scoring and Implementation of the Recommendations . . . 21
5.1.1 Mindexer v0.2.0 Recommends Negatively Performing Indexes 21
5.1.2 Imprecision in Mindexer's Cardinality Estimates . . . 22
5.1.3 Mindexer Variably Recommends Indexes of Differing Performance . . . 24
5.2 Additional Results . . . 25

**6 Analysis of Results** **28**
6.1 Mindexer v0.2.0 Recommends Negatively Performing Indexes . . . 28
6.2 Imprecision in Mindexer's Cardinality Estimates . . . 29

6.3 Mindexer Variably Recommends Indexes of Differing Performance . . 30

**7 Conclusion** **32**

**8 Future Work** **33**
8.1 Performing Additional Sampling Tests . . . . . . . . . . . . . . . . . . 33
8.2 Including Modifying Queries . . . . . . . . . . . . . . . . . . . . . . . 34
8.3 Implementing the Equality, Sort, Range Rule . . . . . . . . . . . . . 35
8.4 Expanding the Maximum Field Size . . . . . . . . . . . . . . . . . . . 35

**Bibliography** **37**

# List of Figures

4.1 LinkBench performance for Mindexer v0.2.0's first set of index recommendations. Evaluated using the final method excluding steps 0, 1, 3, 8, 9, and 10. . . . 18
4.2 LinkBench performance for Mindexer v0.2.0's second set of index recommendations. Evaluated using the final method excluding steps 8, 9, and 10. . . . 18

5.1 Mindexer Modified v3 scores for the covering index (standard queries) 23
5.2 Mindexer Modified v3 scores for the covering index (outlier queries) . 23
5.3 LinkBench performance for each recommended index . . . . 25
5.4 LinkBench performance for the covering index . . . . 25
5.5 Mindexer v0.3.0 performance for each recommended index . . . . 27
5.6 Mindexer v0.3.0 performance for the covering index . . . . 27

# List of Tables

4.1 Mindexer v0.2.0's recommended indexes . . . 19

5.1 Mindexer v0.2.0's relative scores for its index recommendations . . . 22
5.2 Mindexer v0.3.0's recommended indexes . . . 24
5.3 Mindexer Modified v1's recommended indexes . . . 26
5.4 Mindexer Modified v2's recommended indexes . . . 26

8.1 Mindexer's index candidate set sizes . . . 36

# Glossary

**Mindexer v0.2.0** The original version of Mindexer used at the start of this study.

**Mindexer Modified v1** Mindexer v0.2.0 modified with the covering index added to the candidate set.

**Mindexer Modified v2** Mindexer Modified v1 limited to three candidate indexes, including the covering index.

**Mindexer Modified v3** Mindexer v0.2.0 modified to use 10 sample estimators that print out the estimate scores of just the covering index.

**Mindexer v0.3.0** The updated version of Mindexer used at the end of this study, which includes fixes that remove the recommendation of negatively performing indexes, filters out indexes that contain the "_id" field, and applies the sort bonus correctly.

# Chapter 1

# Introduction

Databases are widely used across many major businesses as a key storage system, and the use of indexes within these systems enables the performance of queries to be improved. As a result, it is important that systems use viable indexes that meaningfully and efficiently improve the performance of the database. To help with this problem, MongoDB created an experimental tool called Mindexer, which gathers performance data of queries run by a sample of a workload to recommend indexes for the database in use [1]. Ideally, the recommendations here are the best-case, but these fall within systematic limitations, and its accuracy is to be determined. It is therefore important that we assess just how well Mindexer performs against real-world workloads, especially one that already has a defined developer-recommended index. The workload selected for this study is called LinkBench, modified by the University of Sydney's Database Research group, which is a benchmark designed to emulate workloads similar to Facebook's production environment [2].

The aim of this study is to investigate the results produced by Mindexer, and evaluate its effectiveness across multiple tests. We start by assuming it is indeed producing well-performing indexes (within its known limitations), but then progressively drill down to identify where it is falling short. Performance is evaluated on a local machine in conditions that remain controlled within each run, but oddities do still appear. The initial hypothesis here is that Mindexer does not reliably produce index recommendations that improve database performance. Further tests reveal the complex nature of its results, and how over increasingly more controlled conditions, its inaccuracy becomes easier to identify. After assessing these interesting results, we find and fix a few issues that Mindexer has when generating and scoring candidate indexes, improving its consistency and accuracy. We also identify some remaining flaws relating to its scoring system, and investigate future work to improve it.

This study resulted in two major contributions, the first of which defines the approach used to evaluate the Mindexer recommender, achieved after various tests and fixes, by comparing the performance of what is recommended to what is obtained by a known complex index that is beneficial (the covering index). The second major contribution identifies several weaknesses in the design of Mindexer's scoring system and the implementation of its recommendation algorithm. This is divided into three sub-contributions, which are a fix for a bug in the implementation of its recommendation algorithm where it recommended known negatively performing indexes, (2) an identification of a flaw in the design of its scoring system that shows an imprecision in its cardinality estimates, and (3) the identification of a weakness in the design of its scoring system that resulted in it variably recommending indexes of differing performance, allowing multiple unequal indexes to share the same final score.

# Chapter 2

# Background

MongoDB is a widely used NoSQL database management system (DBMS) that utilises collections of flexible Binary JavaScript Object Notation (BSON) [3] documents to store data, as an alternative to tables and rows as used in a classic relational DBMS [4]. This approach allows users to store various data types and nested structures, such as arrays and other documents, in each document, improving flexibility as the schema (database structure) is dynamic. Due to how it stores its documents, distribution across multiple servers is simple, making it easy to create highly scalable systems [4].

Creating indexes is a common method of improving performance in large databases, but they must be selected carefully, as the wrong choice can also negatively impact query performance. An index stores a collection with entries containing its designated search key(s) and pointers to the location of its corresponding values in the database. As such, a query that is able to utilise an index can avoid performing a full collection scan, as it has a quick reference to the matching values. The selection of indexes is also a balance between performance and memory usage, as larger indexes that cover more fields will typically take up more space as there will be more columns and entries due to the increased granularity. The performance of an index is highly dependent on the types of queries in use by the workload, as operations that modify the collection will require the index to be updated. Hence, it is important that indexes are chosen on fields that result in fast read performance, while considering the impact of updates.

Mindexer is an experimental tool created by MongoDB, designed to examine workloads and recommend indexes to improve its performance [1]. It does this by using the information stored in the MongoDB Database Profiler (which monitors queries and logs their results to a separate collection [5]) to extract the fields in use by the workload to generate a set of candidate indexes to be tested. These candidates are based on the various permutations of fields used in the queries, and are currently

limited to 3 fields per index. Then, Mindexer creates a sample of the database collection used, such that it has some data to score the indexes on [1]. Each candidate index is then scored against each query as logged, by first checking if it intersects with the query, meaning it would be utilised by the query. If this is the case, its score is calculated using cardinality estimation, which is based on the number of records returned by the sample database for the intersection between the query and the index. This score estimate is scaled up to match the true database size, and is then used to calculate a benefit value which also has a bonus added if the index can be used to sort the results within the query. The benefits for each query are then summed up to produce a final benefit estimate for the index. To select the best candidate index, Mindexer uses a greedy algorithm to pick the index with the highest benefit, and recommends it as its first choice. The tool then recalculates the benefit values for the remaining candidates, to see if another index could further improve performance if implemented alongside the previous recommendations. If there are indexes fitting these criteria, they are recommended as well.

The query optimiser in a DBMS is responsible for deciding how a query will be executed, and it does this by generating different plans that perform the necessary operations in different ways. To select the most performant operations, it utilises cardinality estimation, which guides its choices by estimating the size of the result sets from the generated sub-plans without ever executing the query [6]. There are various different techniques for performing cardinality estimation, but three main types stand out: histogram-based, sampling-based, and machine-learning-based. Histogram-based and sampling-based methods are more traditional but are widely applied and utilise simplified assumptions. Machine-learning-based methods are more novel and rely on larger sets of data to produce more accurate results [6]. For Mindexer, sampling-based cardinality estimation is utilised, as it allows for faster recommendations.

LinkBench is a database workload benchmark created to emulate the database and requests as used by Facebook in their production environment [2]. This is essentially a social network graph, where nodes represent people, posts, etc., and links represent the relationships between the nodes, such as friendship between people, a person liking a post, and more. Each of these nodes and links can also contain additional information, and are altogether stored in three distinct collections: one for nodes, one for links, and one for link counts. The LinkBench benchmark first loads up the social graph by creating the various nodes and links as specified by the configuration files. The number of links created follows a power-law distribution, such that most nodes have a maximum number of links, but some nodes have many more links [2]. The benchmark then moves on to performing requests on the graph, which mimics

the way Facebook's production system calls its database. Here, a combination of read and write operations are performed, with some nodes accessed frequently, and others accessed rarely. At the end of this request phase, LinkBench reports some statistics that can be used to evaluate the overall performance of the workload on the database as configured [2].

# Chapter 3

# Literature Review

This chapter summarises various related works in this research space, with a major focus on studies surrounding the different methods of index selection across major database systems.

## 3.1 Database Performance

Databases are now widely used in the industry, and their importance and continued use have resulted in a considerable amount of research into how they perform, where performance can be improved, as well as various methods and technologies to help with performance. According to C. Cioloca, M. Georgescu, et al., "performance is one of the most important metrics" when describing a project's success. Hence, it is crucial that development considers performance as early as possible [7]. They explore this notion in *Increasing Database Performance using Indexes*, where their findings show that the entire system's performance is improved when indexes are used. Tests were performed on two different types of indexes: clustered and non-clustered. Here, a clustered index is defined as one that physically contains the ordered table data, while a non-clustered index is when there is no specific order to the key values [7]. Due to the nature of clustered indexes reordering the data rows, only one of these is allowed per table, but many non-clustered indexes can be made as they do not physically alter the ordering in the table. With this limitation in mind, it is also important to note that clustered indexes need to be reordered whenever the index or column is updated, which is why it is not recommended for columns that are frequently updated [7]. The authors note that while the values in clustered indexes point to a specific data row location, this is not always the case with non-clustered indexes, as its values can point to the keys in a clustered index. Hence, clustered indexes should be created before the non-clustered indexes, and this also means that when a clustered index is rebuilt, the non-clustered index will also have to be rebuilt [7]. Due to these behaviours, they recommend clustered indexes to

be used when retrieving large amounts of data, and non-clustered indexes for smaller requests. This does seem to be counter to accepted techniques and is a limitation of this paper, as the general belief is to choose clustered indexes for range-based accesses, because the size of the data returned does not necessarily mean the data is close together. Their final conclusions point out that it is important to choose the right number of indexes, as too many can impact performance negatively, so indexes that are infrequently used should be removed or merged in favour of indexes that are consistently used [7].

## 3.2 OLAP Index Selection

One of the earlier studies presenting index selection algorithms saw its focus placed on online analytical processing (OLAP) systems, which is different from the online transaction processing (OLTP) system used with LinkBench. In *Index Selection for OLAP*, Gupta et al. describe OLAP as an interactive decision-support process, an important application of database systems. The data contained in these systems are typically presented as a “data cube” which is multidimensional in nature [8]. As such, a common method to reduce the execution time of queries is to precompute them as summary tables, and then build indexes on top of them. This two-step process uses a trial-and-error approach to divide the space used for the summary tables and for the indexes, by picking the tables first, and then selecting the indexes on them, which typically performs rather poorly [8]. The authors note that performance can be improved by selecting the tables and indexes together, since they consume the same space. As a result, various algorithms with different levels of complexity were presented, which automate the selection of the summary tables and indexes. The algorithms built consisted of the r-Greedy algorithm(s) and the Inner-Level Greedy algorithm, tested using a cost model devised by the authors as well [8]. After extensive tests using the cost model on these algorithms, the results showed that “higher complexities have better performance bounds”, but this diminishes as the complexity increases. They found that even a moderately complex algorithm could perform close to the optimal one, and concluded that the precomputation of summary tables and indexes should be integrated into one step [8]. It is worth noting that this paper is limited as it focuses on selecting indexes for use in data cube calculations, which is quite different to selecting indexes for SQL queries. As well, its focus on OLAP systems means that it is not as widely applicable to OLTP systems like LinkBench, which access fewer documents but across more fields, and can contain modifying queries.

## 3.3 Automated RDBMS Index Selection

The earliest paper used in this review sets its sights on relational database management systems (RDBMS), which are the most popular of the database systems. Here, indexes are used to quickly provide access to data, but do complicate update operations as the tuples and the indexes have to be updated [9]. As stated in *Adaptive and Automated Index Selection in RDBMS*, choosing the right attributes to index is quite difficult, as this is influenced by how the database is used, but also the general characteristics of the database and operating system. This study by Frank et al. improves upon other studies, as the tool it describes automatically derives the workload in use by analysing usage, instead of needing it to be manually specified. Their tool generates an index benefit graph by asking the optimiser for all of the indexes it would choose for a given query. These results are stored and are used by the tool to extract the best possible index choices based on the total workload [9]. A limitation here is that the tool was run on a simulation of the optimiser, but when testing the results on a real database, there were some very clear performance improvements when the database had a large gap in terms of how it performs in the best-case against the worst-case. Another limitation is that the tool does not consider compound (multi-field) indexes, but this was normal for its time, as no other tools did this either [9]. The authors concluded their study with the notion that their tool is successful if the optimiser in use is able to estimate the costs of index sets, and its results are usually quite close to the optimal solution when this is the case. Ideally, a future study would extend this experiment by using an actual optimiser [9]. There are some similarities between the tool presented in this paper and how Mindexer selects its indexes, as both break down the assessment of preferred indexes on a query-by-query basis, but Mindexer is more advanced in that it can handle multi-field indexes, up to a limit.

## 3.4 Relational Database Index Selection

Choenni et al. explore the idea of developing a tool that supports the physical design of relational databases by noting that it is reliant on understanding index selection. *Index Selection in Relational Databases* dives into the selection of primary indexes and their relation to secondary indexes, due to the majority of studies at the time paying more attention to just secondary indexes. In this paper, the authors state that indexes can be considered as “auxiliary files” which allow for the retrieval of tuples that satisfy certain predicates without examining the entire relation. Indexes also have to be updated when the database is updated to maintain consistency, hence it speeds up retrieval but slows down maintenance [10]. This paper was different

from others, which typically focused on just choosing the primary or secondary index, while the authors here selected both, and examined the differences in results depending on the order of selection. They developed a cost model to evaluate the costs of just a primary index, just a secondary index, and then the combined costs. Then two analytical evaluations were performed, where the first involved selecting the primary index before the secondary index, and the second was the reverse [10]. The authors found that the secondary index set rarely impacts the gains produced by that of the primary index, so the selection of secondary indexes could be done independently to cover areas that were not serviced by the primary indexes. Their conclusion showed that it was better to select the primary indexes first, but this selection should be done carefully to prevent sub-optimal configurations [10]. The methods used in this paper echo much of how Mindexer currently operates, with it also assessing the benefits of candidate indexes against one another using a cost model, as well as starting off with the base assumption of suggesting a primary index first.

## 3.5 Cost-Driven Index Selection

*An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server* focuses on the building of an industry tool that can automate choosing indexes for SQL databases, based on a given workload of queries. This paper builds upon the past three reviewed index selection papers, but is notably different in its approach. Frank et al.'s study [9] is stated to not be scalable to large workloads, while their methods are distinct from that of Choenni et al. [10]. They also mention the work done by Gupta et al. [8], and how it is limited by assuming that the use of one index does not impact another. The authors first start by emphasising that an important part of being a database administrator is choosing a database design that fits the workload of the system, but also selecting appropriate indexes [11]. The goal of the paper is to "pick a set of indexes that is suitable for a given database and workload", and that there is no restriction on the types of indexes used. The tool presented by Chaudhuri and Narasaya iteratively selects candidate indexes that gain in complexity, and progressively eliminates the indexes that show low benefits against the workload. Indexes that do well are considered in the next iteration, alongside new multi-column indexes that introduce additional attributes to the already good indexes [11]. They use an algorithm to reduce the number of optimiser calls made when evaluating the performance of the index on that workload, by testing a limited number in a simulated database. As there can be a large number of candidate indexes, the authors reduced the candidate pool by using a greedy algorithm to get the best configuration for a specific query. As a result of this study and their

novel implementations, they were able to improve efficiency by up to a factor of 10, while maintaining a similar index selection quality. This tool (named AutoAdmin) was then implemented in Microsoft SQL Server 7.0 [11]. This paper bears many similarities to Mindexer and how it selects indexes, as the tool uses the provided workload to generate and assess indexes in a, but performs an eliminate-and-iterate approach where Mindexer simply scores all of them. Their tool is limited by how it greedily removes candidates based on one query, whereas Mindexer does not make that assumption as other queries could have benefited from an eliminated candidate.

## 3.6 The Hardness of Index Selection

In this paper, Chaudhuri et al. return with a new focus on recommending indexes that maximise the benefit on the workload when provided with a storage constraint. *Index Selection for Databases: A Hardness Study and a Principled Heuristic Solution* can be seen as complementary to that of their prior paper [11], since this dials in on the configuration selection aspect. This paper is different from Choenni et al.'s [10] study, which utilised an explicit cost model, whereas here an optimiser estimate is used. Again, the work done here differs from that of Gupta et al. [8], which assumes that the selection of one index does not impact another, but this time they mention that there are similarities in how they observe the benefit against the workload, instead of just the total cost. "Index selection can be viewed as an optimisation problem" states the authors, as it is a balance of finding the best indexes that fit the given constraints [12]. Their prior paper [11] suffered from not being able to guarantee their solution's quality, and it had no analysis of the hardness of index selection. In this study, they show that picking the right set of indexes, regardless of whether it is clustered or non-clustered, is both NP-hard and hard to approximate. So, they developed a heuristic solution to select indexes by treating it like the knapsack problem, by assigning benefits to each index candidate [12]. The authors present a configuration selection algorithm that addresses the issues with the previous studies and problems with the hardness, where indexes are given a benefit value that applies to the entire workload. Then, they greedily select the best indexes while remaining within the constraints of the system. Their tests showed that the performance of their algorithm is generally rather comparable to that of paper [11], but this new algorithm makes significantly fewer optimiser calls for larger workloads, which is beneficial for running times. However, it also has a slight decrease in quality when this happens, but the results were enough to prove that their new algorithm is more scalable. Also, due to the new tests, the algorithm has guaranteed the quality of its results [12]. As this tool builds upon their previous paper, it inherits the similarities to Mindexer's implementation, but with it now

also assigning benefits to each index candidate, it effectively becomes an alternative to Mindexer which uses the workload optimiser instead of cardinality estimation through database sampling.

## 3.7 Evaluating Index Selection Algorithms

A more recent paper sees Kossmann, Halfpap, et al. evaluate various different index selection algorithms. As stated in the paper, the use of indexes is essential if efficient processing of database workloads is desired. The index selection problem has many solutions, ranging from metadata-based heuristics to multi-step algorithms, all the way to algorithms that actually provide optimal results. However, the challenges associated with this are how to accurately determine an index's effect on the workload in use while also taking into account the other indexes present, and the sheer amount of candidate indexes that are available for large workloads [13]. *An Experimental Evaluation of Index Selection Algorithms* tackles this issue by examining eight existing index selection algorithms across different dimensions, e.g. quality of solutions, multi-column support, complexity, and more. These are done on three workloads, the Join Order Benchmark (JOB), TPC-H, and TPC-DS, which are all described as challenging workloads for the algorithms in use. The algorithms selected were Drop, AutoAdmin (from paper [11]), DB2Advis, Relaxation, CoPhy, Dexter, Extend, and Database Tuning Advisor (DTA). To perform their evaluations, Kossmann, Halfpap, et al. developed a publicly available platform to ensure their results are reproducible, their tests are extendible to other algorithms, workloads, and systems, but also to automate the process [13]. Across the eight algorithms chosen, there are some notable differences in how they select indexes, as half of the algorithms were designed to minimise cost, while the other half were built to balance costs against storage. The majority of algorithms stop execution when a storage limit is hit, but two stop when an index limit is hit, and one stops once a savings threshold has been achieved. Nearly all algorithms support multi-column indexes, and all of them consider the interactions between indexes [13]. The three workloads utilised by the authors varied in complexity, where JOB is a real-world dataset and has the least multi-column candidates, while TPC-H and TPC-DS are synthetic benchmarks, with TPC-H being simpler than TPC-DS in terms of scale [13]. Their evaluation platform took advantage of PostgreSQL and HypoPG to estimate hypothetical index costs, and was built on Python 3. The platform performs the setup, generates queries, evaluates the algorithms, and summarises the results. Once the tests were carried out, their findings showed that the different algorithms perform best depending on the workload and restrictions [13]. Generally, AutoAdmin and DB2Advis are recommended for when fast solutions are needed, while Relaxation is

better for when runtime is not a concern. Smaller problems favour CoPhy's optimal results, while Dexter is simple to use. Drop is the easiest to implement, but Extend and DTA give the best balance of runtime against solution quality. The authors concluded by stating that efficient index selection is still quite difficult, as the problem results in wildly differing performance depending on the scenario. However, due to their introduced platform, further evaluations by the greater public can be easily carried out [13]. As this tool builds upon their previous study, it inherits the similarities to Mindexer's implementation, but with it now also assigning benefits to each index candidate, it effectively becomes an alternative to Mindexer which uses the workload optimiser instead of cardinality estimation through database sampling.

## 3.8 Index Selection using Reinforcement Learning

In *SWIRL: Selection of Workload-aware Indexes using Reinforcement Learning*, a new algorithm is introduced and compared to those performed by Kossmann, Halfpap, et al. in paper [13]. The authors begin by noting that current index selection algorithms are not fast or competitive enough for complex workloads. With their approach, SWIRL, reinforcement learning (RL) is used to offer both of these advantages [14]. As noted in the paper, more than 75% of all databases are expected to be run in the cloud by 2022, which means effective physical database design has been shifted to be the responsibility of the cloud vendors. In these cloud software scenarios, thousands of users will run workloads on similar schemas as they are predefined, so the problems for index selection are similar. With current index selection approaches, prior knowledge is not taken into account, and that can help improve performance, Careful RL can take advantage of a large amount of existing training data to almost instantly infer the best indexes [14]. The SWIRL tool works by dividing the RL index selection process into three phases: a preparation phase where the index candidates are chosen and the training workload is prepared, a training phase where the tool learns which indexes are valuable and their interactions, and finally an application phase where the tool determines the best index using the trained model. Kossmann et al. implemented the tool in Python3 and tested it on the same evaluation framework as in paper [13], with PostgreSQL and HypoPG. Stable Baselines was used for the RL algorithm, while Gensim was needed for the workload representation [14]. Their tests saw the tool compared against three of the algorithms from paper [13]: AutoAdmin (the most well-trained, and is from paper [11]), DB2Advis (the fastest), and Extend (the best). DRLinda, another RL-based selection approach was also included in this comparison. SWIRL performed really well in these tests, outperforming every other tool by quite a large margin in terms of speed across various workload sizes. It generally does well to match the other

algorithms for the relative workload cost, and routinely beats DRLinda in this scenario. A noticeable misstep is seen when run on JOB, where it has one of the lowest workload costs, but it performs pretty well in the other two workloads, and in all three scenarios it is still the fastest [14]. Kossmann et al, conclude that SWIRL is best applied when many index selection problems are to be solved, as it increases its training duration in order to reduce its solution runtime. Future studies could see the tool be improved to reduce training times and integrate more of the physical database design [14]. The SWIRL tool is quite different to Mindexer, as it utilises RL, while Mindexer uses simple scoring models and does not use a trained model. Mindexer could benefit from RL, but the increased running times may be counter to its design prioritising speeds from sampling the database.

## 3.9 Scalable Index Tuning

In this study, Siddiqui et al. build upon the proposals suggested in paper [11] to help reduce the number of optimiser calls utilised by index tuning algorithms. *DISTILL: Low-Overhead Data-Driven Techniques for Filtering and Costing Indexes for Scalable Index Tuning* discusses how resource-intensive and time-consuming index tuning can be in many current database systems, but it is still used as a means of selecting appropriate indexes to improve workload performance [15]. The current methodology for tuning indexes is to first generate a set of candidates based on the queries utilised by the workload, and then drill down to find the best configuration of indexes using configuration enumeration. This performs what-if calls to the optimiser to estimate the cost of the query if the index was applied, without actually creating the index. Here, Siddiqui et al. mention that the technique in [11] helps reduce the search space of candidate indexes, but many optimiser calls will still have to be performed [15]. The first of their contributions, Index Filter, is a pre-trained model that eliminates candidate indexes that would not result in performance improvements, which are typically the majority of candidates. Index Filter does this by searching for patterns it has identified and learned from previous workloads. The authors have also developed a means to replace many of the optimiser calls that are used to estimate costs with Index Cost Models, which is a model trained on the current workload to predict the costs of the query with the index applied. This can be done on the current workload as there is less variation in the costs compared to the indexes [15]. Together, these developments are implemented in the DISTILL prototype, which reuses the proposed techniques in [11], such as its candidate index generation and greedy search. Siddiqui et al. then compared DISTILL to the Database Tuning Advisor (DTA) on various benchmarks and workloads, showing that DISTILL greatly reduces tuning times while giving similar quality recommendations

as DTA [15]. Mindexer has a few similarities with this prototype, as it also uses a cost model to predict index performance, but is simpler in its implementation as it utilises predefined values. The methods used to eliminate candidate indexes could be useful in Mindexer, especially as it utilises a pre-trained model.

# Chapter 4

# Methodology

This chapter describes the novel approach we developed to evaluate an index recommender, such as the different versions of Mindexer, on a real workload, such as LinkBench. A similar approach could be adapted and applied for evaluating index recommenders on other platforms or workloads.

## 4.1 Contribution: Approach to Evaluating the Recommender

The essence of our proposed approach for evaluating an index recommender is to apply the recommender to a workload, and then to measure the actual performance of that workload with several different indexes. We compare the running time of the workload with the recommended index to a control (the time with no index added), and additionally to the time with an index suggested by the developers of the workload. In the case of a recommender such as Mindexer, which recommends a list of indexes to be added sequentially, we measure the running time with just the top recommendation, then the time with the top 2 recommendations, etc. As we are testing on the LinkBench workload, the developer-recommended index is a covering index designed for its implementation on MongoDB. The rest of this section describes the application of this evaluation approach in detail.

To evaluate the performance of the recommendations from Mindexer, a local MongoDB server had to be set up first, with the LinkBench tool configured to connect to it. The LinkBench tool was then run in its loader mode to set up the relevant collections that it needed for its tests on the MongoDB server. Then, the MongoDB query profiler was set to the most detailed logging mode and the LinkBench tool was run in its requester mode.

After the first run was completed, the profiler was turned off and the Mindexer tool was run with a connection to the MongoDB server and the LinkBench collection to be used such that it could create candidate indexes and evaluate its projected performance against the workload. Then, its top index recommendations were applied to the database and the LinkBench requester was run against them, such that the true impact of each index on the workload's performance could be assessed.

For completeness, a LinkBench requester run with the suggested covering index created on the database was performed, so that its performance could be compared to the index recommendations.

In order to ensure consistency within each test, a few controls were utilised. These include closing all other applications on the host machine and resetting the database between and during each run. As such, the final testing process for Mindexer's index recommendations on LinkBench followed this set of steps:

0. Close all applications on the host machine
1. Clean the database and create the collections using the LinkBench loader
2. Assess the performance with no indexes using the LinkBench requester
3. Delete the collections and regenerate them using the LinkBench loader
4. Apply the (first) index recommendation(s) to the database
5. Assess the performance with the index(es) using the LinkBench requester
6. Repeat steps 3-5 with each additional index, where prior indexes are applied in order
7. Repeat steps 1-6 as many times as necessary
8. Delete the collections and regenerate them using the LinkBench loader
9. Apply the LinkBench recommended covering index to the database
10. Assess the performance with the covering index using the LinkBench requester

Steps 1 to 7 were implemented using two scripts that were specifically made to automate the process, with intervention (by the tester) only needed to ensure that each script stayed up-to-date with the other.

It should be noted that when Mindexer suggests a set of indexes, the performance gains stated are relative to each other, hence the second index recommended is reliant on the first index recommendation being applied first. This was followed in our tests as outlined in the steps above. For the purposes of streamlining this study, Mindexer was only used on the "linktable" collection utilised by LinkBench, which was the

largest of the three created collections and had the majority of complex queries. This is due to Mindexer being limited as it can only assess the queries performed on one collection at a time. Another limitation of Mindexer is that it only scores indexes against queries that do not modify the dataset, which means that 63 out of the 450 queries performed by LinkBench are excluded from its scoring algorithm. As well, Mindexer is unable to recommend the covering index due to its field-length restrictions, as only indexes of up to three fields will be considered.

In the LinkBench workload, the linktable collection's queries mimic real-world queries on the Facebook social network graph, and fall into two major categories (or patterns). The majority of fetch queries (245 out of 387) are simple equality filters on the specific '_id' field, with only 1 result returned. The rest of the fetch queries perform an equality filter on three of the seven fields in the linktable collection, range filters and sorts on the 'time' field, and exclude the '_id' field from the final output, which can vary from a few results to above a thousand results.

## 4.2 Further Insights

As Mindexer is built to specifically work with MongoDB, the usage of MongoDB as the database server was effectively a requirement. LinkBench was chosen as the workload to assess Mindexer on due to its real-world applicability, with it designed to emulate the Facebook social network graph. This workload benchmark has also never been used with Mindexer, making this a novel experiment.

The proposed method to evaluate the performance of the recommender was developed incrementally, and started off without any controls except for the host machine remaining the same. Here, Mindexer v0.2.0's first set of index recommendations was used, of which there were 51 results. In order to reasonably test performance, only the top 5 recommendations were applied. As seen in Figure 4.1, the starting performance of each run differed quite greatly, especially in the case of Run 2. The indexes also had varying impacts on the performance of the requester, with Run 3 seeing performance get worse when the fifth index was applied.

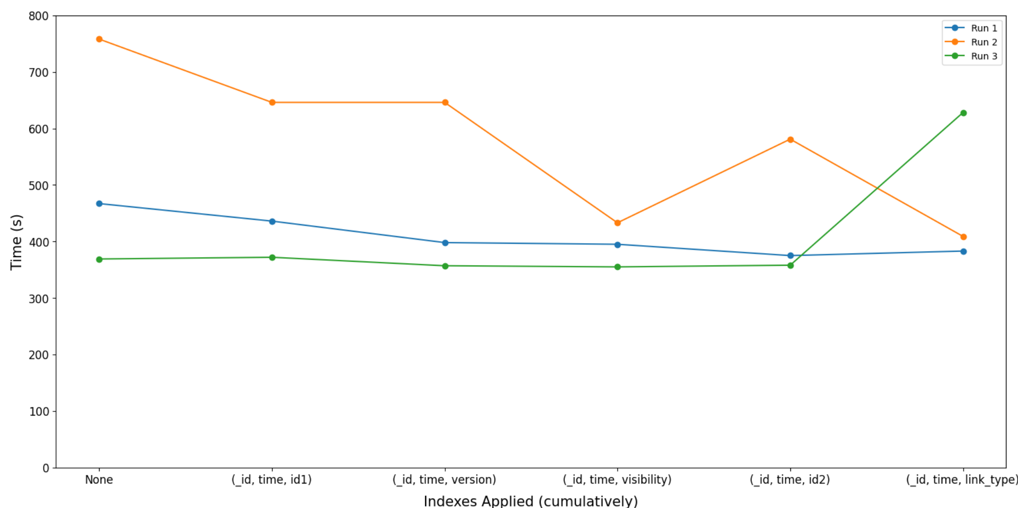



Figure 4.1: LinkBench performance for Mindexer v0.2.0's first set of index recommendations. Evaluated using the final method excluding steps 0, 1, 3, 8, 9, and 10.

In order to achieve a more reliable evaluation of performance, the controls mentioned above (closing all applications and resetting the database between and during runs) were introduced. Additionally, Mindexer v0.2.0 was run again prior to this test, where this time it produced 2 index recommendations, and both were applied. As seen in Figure 4.2, the new controls did not have much of an impact on the general starting performance of the workload, but did bring consistency to the performance within the run, as the impact of each index was consistent. Here, the performance within each run carried over to the first index, but the second index brought large improvements to the workload's performance across all 5 runs.

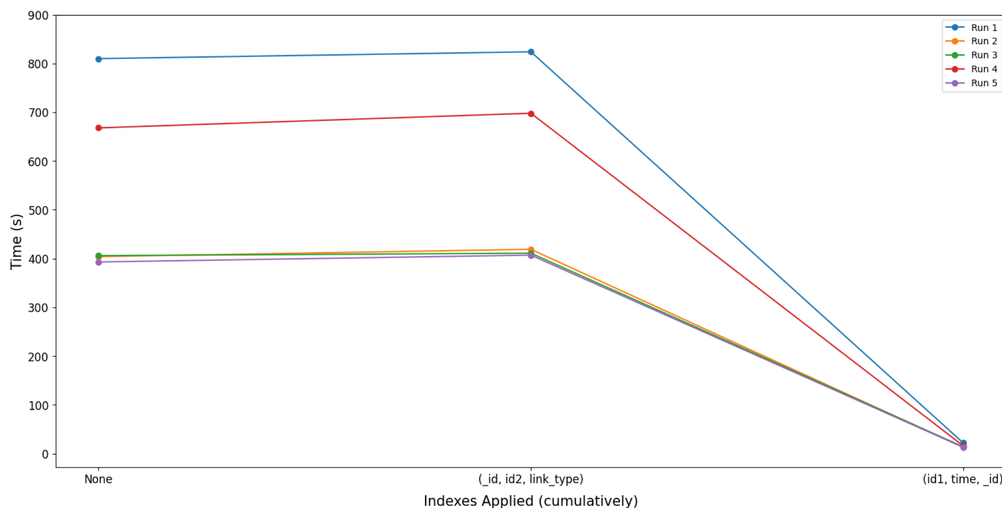



Figure 4.2: LinkBench performance for Mindexer v0.2.0's second set of index recommendations. Evaluated using the final method excluding steps 8, 9, and 10.

The first two Mindexer v0.2.0 index recommendation outputs, used in Figure 4.1 and Figure 4.2 respectively, are shown in Table 4.1. It should be noted that the performance of the covering index was not tested in either of these initial evaluations, as the evaluation approach was still being developed. The full method was applied in later performance evaluations, with the scripts utilised as mentioned in Section 4.1.

| **Run 1** | **Run 2** |
|---|---|
| (_id, time, id1)<br>(_id, time, version)<br>(_id, time, visibility)<br>(_id, time, id2)<br>(_id, time, link_type)<br>... | (_id, id2, link_type)<br>(id1, time, _id) |

Table 4.1: Mindexer v0.2.0's recommended indexes

Due to these initial evaluations, an investigation into the actual scores as calculated by Mindexer was performed, where a slight modification to the code made it such that the recommendations also showed the score it was assigned.

Once enough results were gathered from the initial evaluations, we pivoted to testing the underlying algorithm of Mindexer. To do this, the covering index (as recommended by the LinkBench developers) was placed alongside the generated candidate indexes (in Mindexer Modified v1), and the output produced was observed. Further tests saw larger changes that included turning off the candidate selection process entirely (in Mindexer Modified v2), and we identified inconsistencies in the results due to the sample size taken from the database.

Mindexer samples a random 0.1% of the provided workload, which meant that its calculations were based on differing values in each run. We performed a large test that involved multiple runs of Mindexer with varying sample ratios, using Mindexer Modified v3 designed to test just the scoring system of Mindexer. Each run in this test instantiated the sample database 10 times to get a decent number of data points, to be used specifically on the covering index.

The following four database sources (sample ratios) were utilised in this test:

- The control: One run that used the full dataset (sample ratio of 1)
- The original: Three runs that used the initial Mindexer configuration (sample ratio of 0.001)

  **Note:** while this sample ratio aims to sample 0.1% of the database, there are not enough entries to meet the minimum required by Mindexer, hence it instead samples some 1000 entries, which comes to about 0.275% (sample ratio of 0.0027). For simplicity, it will be referred to as the run with a sample ratio of 0.001, as that was the number provided to the Mindexer configuration.

- Three runs that used a sample ratio of 0.01
- Three runs that used a sample ratio of 0.1

After these tests, improvements were made to the Mindexer code to solidify consistent behaviour when running with a sample ratio of 1, which included a bug fix to its recommender as it was recommending indexes that it knew had no further benefits. Further tests were then performed to dive deeper and gain an understanding of Mindexer's scoring system, with its candidate generation reactivated to get more coverage. We made use of its built-in performance tester (where it runs the given workload before and after applying its index recommendations), and rotated the tests across the same 4 sample ratios as listed above. From these tests, some bugs were identified in the code which were subsequently fixed, namely one that resulted in the candidate pool being flooded with candidates that would not have much of an impact on the actual workload, and another where the calculation of a "sorting bonus" was given to indexes incorrectly, which also lowered the scores of the covering index.

With the identified bugs rectified, Mindexer v0.3.0 was used to generate its new set of index recommendations. These indexes were evaluated using our finalised approach, with five runs performed on the LinkBench workload using the automated scripts and the covering index recommended by the LinkBench developers included for completeness as well. A similar evaluation was carried out on a customised version of Mindexer with candidate generation turned off, with 15 tests run on its built-in performance tester, to see if there were differences in the relative running times of the indexes when using just non-modifying (fetch) queries.

# Chapter 5

# Results

This chapter identifies the various weaknesses we discovered while evaluating the Mindexer recommender, as well as any additional results obtained during our investigations. The results from the major contribution are divided into three sub-contributions, each identifying a unique flaw with Mindexer.

## 5.1 Contribution: Identification of the Weaknesses in the Design of the Scoring and Implementation of the Recommendations

### 5.1.1 Mindexer v0.2.0 Recommends Negatively Performing Indexes

Mindexer v0.2.0 was run five times on the LinkBench workload, with a slight modification to the code made such that the recommendations also showed the score it was assigned. These results are listed in Table 5.1, limited to the first 11 recommendations. In Mindexer, the score of each recommended index is relative to the previous index recommendation, which means it represents the performance benefit of adding this index to all previously recommended ones. Here, it was discovered that Mindexer was recommending indexes that it knew had no additional benefit (or would make performance worse), which should not be the case according to Mindexer's design.

| Recommendation Number | Relative Scores | | | | |
|---|---|---|---|---|---|
| | **Run 1** | **Run 2** | **Run 3** | **Run 4** | **Run 5** |
| 1 | 99420158 | 99201290 | 99041380 | 99625647 | 98993927 |
| 2 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 |
| 7 | -2978848 | -2759980 | -2600070 | -3184337 | -2552617 |
| 8 | -2978848 | -2759980 | -2600070 | -3184337 | -2552617 |
| 9 | -2978848 | -2759980 | -2600070 | -3184337 | -2552617 |
| 10 | -2978848 | -2759980 | -2600070 | -3184337 | -2552617 |
| 11 | -2978848 | -2759980 | -2600070 | -3184337 | -2552617 |

Table 5.1: Mindexer v0.2.0's relative scores for its index recommendations

These inconsistent results and recommendations for known negatively performing indexes were later fixed and included as an update to the Mindexer codebase as part of Mindexer v0.3.0. Insights into why this occurred and how they were fixed are discussed in Section 6.1.

### 5.1.2 Imprecision in Mindexer's Cardinality Estimates

Mindexer's estimated index scores were varying between runs, and it was important to discover why this was the case. The hypothesis was that it arose due to a difference in the sample data used to estimate the cardinalities of each query. As such, the scoring was evaluated using Mindexer Modified v3 to observe the differences in estimates when changing the sample ratio in use, from the default of 0.001, up to a ratio of 1, where the full collection is used. These results were then graphed using the median of initial cardinality scoring estimates on the covering index, prior to Mindexer applying a function to calculate the final benefit for it. The graphs were split into two parts, one for the majority of queries, and the other for the high-scoring outliers.

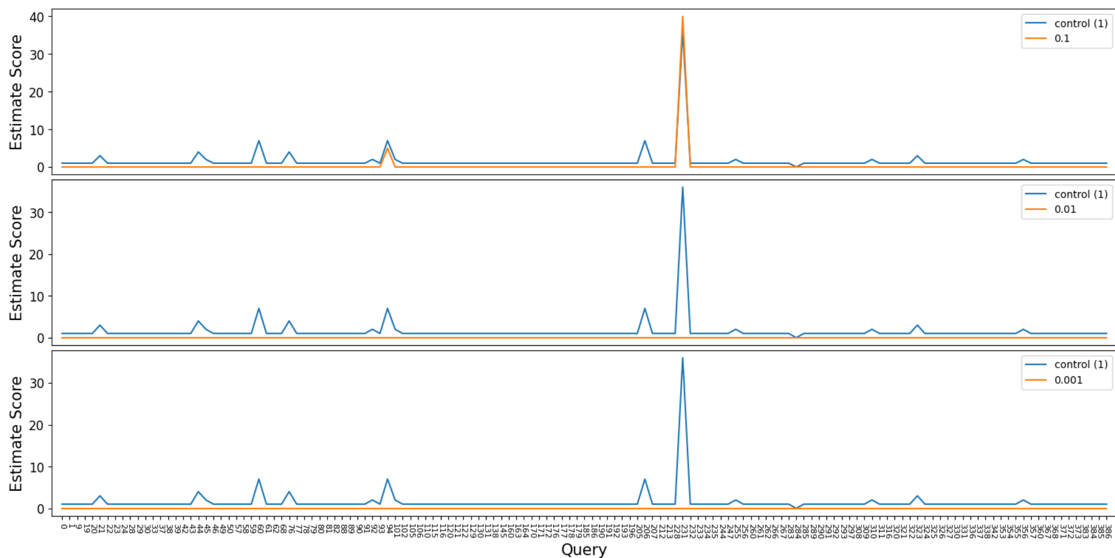



Figure 5.1: Mindexer Modified v3 scores for the covering index (standard queries)

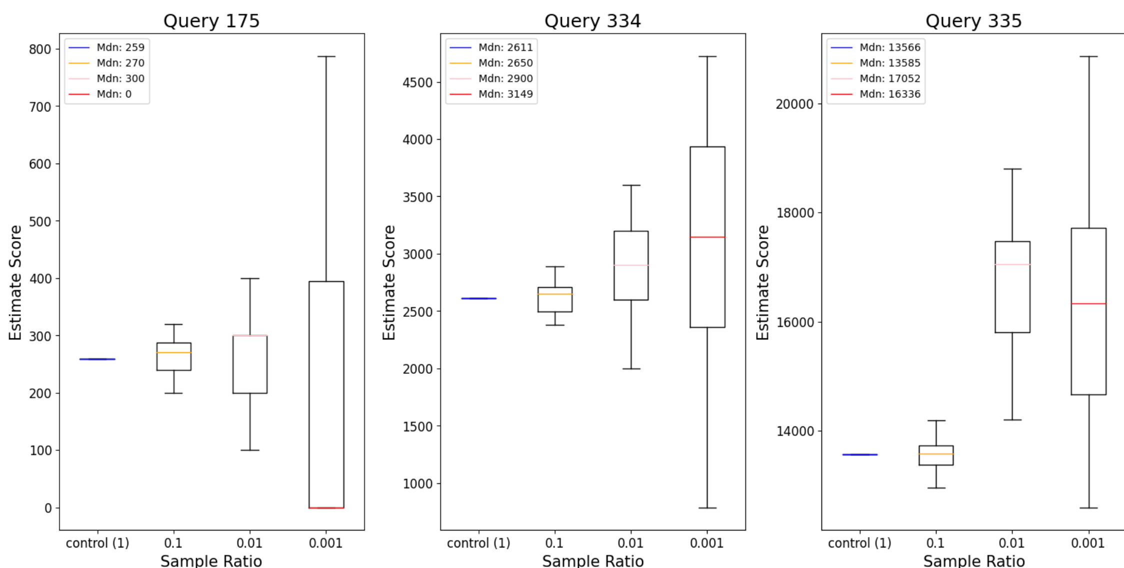



Figure 5.2: Mindexer Modified v3 scores for the covering index (outlier queries)

As seen in Figure 5.1 and Figure 5.2, the results fall in line with expectations, where the larger sample ratios produced estimates increasingly closer to the control (1). For the majority of queries, even a sample ratio of 0.1 rarely produced results that matched the control, as it is only the larger outliers where the differences were easier to notice. It is in these outlier graphs where it can be seen that an increase to a sample ratio of 0.01 brought results closer to the control compared to the original 0.001. However, this observation did not apply to the median scoring of query #335, where the sample ratio of 0.001 more frequently had a closer estimate over 0.01. At a sample ratio of 0.1, the scoring estimates by Mindexer were very close to that of the control, such as in query #335 where the median cardinality estimate is off

by only 0.14%. There is a clear trend showing the range of scores decreases as the sample ratio increases, demonstrating the effect the sample ratio has on the overall accuracy of the cardinality estimates.

An observation from these results is that Mindexer is more precise with the outlier (complex) queries, as its estimates are much closer relatively than with the standard queries. The accuracy of the estimates directly affects the calculated benefit of the index on the query, and further insights into how this impacts the recommendations are discussed in Section 6.2.

### 5.1.3 Mindexer Variably Recommends Indexes of Differing Performance

In these performance tests, it was first observed that Mindexer v0.3.0 now consistently scored the same 4 indexes when run with full candidate generation (and no covering index inserted). The indexes and their initial scores are shown in Table 5.2, which lists the final benefit score Mindexer assigns prior to re-calculating the scores to provide relative scores. While the top 4 were always the same, the final recommendation would differ between runs, as the ordering was non-deterministic. Hence, the four indexes were tested using the final evaluation approach as described in Section 4.1 to see if they were truly equal, by comparing the difference in workload performance with and without the index. As mentioned in the method, the covering index was also tested on LinkBench for completeness, and for reference, that index is (id1, link_type, visibility, time, id2, version, data).

| Index | Median Initial Score |
|---|---|
| (id1, time, link_type) | 58422050 |
| (time, id1, link_type) | 58422050 |
| (link_type, time, id1) | 58422050 |
| (time, link_type, id1) | 58422050 |

Table 5.2: Mindexer v0.3.0's recommended indexes

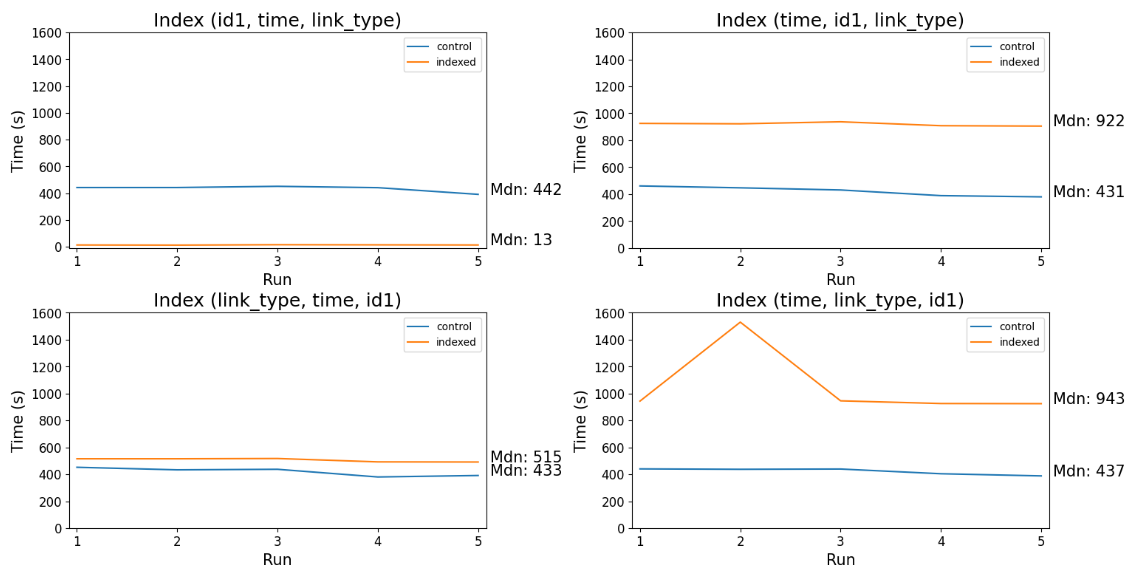



Figure 5.3: LinkBench performance for each recommended index

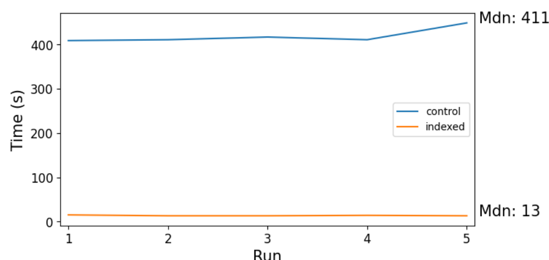



Figure 5.4: LinkBench performance for the covering index

It can be immediately noticed in Figure 5.3 that the indexes were not performing identically, with only one index actually improving performance over the control (no-index) run, significantly. This index, (id1, time, link_type), has its median time taken matching that of the covering index seen in Figure 5.4. The other three indexes were negatively impacting the workload's performance, most notably the two indexes which have "time" as their primary field, which take more than twice as long when compared to the control. This is likely due to many queries having a range filter and sort on this 'time' field, as mentioned in Section 4.1, and the impact of this is further discussed in Section 6.3.

## 5.2 Additional Results

### Mindexer v0.2.0 Ignores a Well-Performing Index

Prior to testing the effect of changing the sample ratio, Mindexer's scoring algorithm was assessed using Mindexer Modified v1. Firstly, the developer-recommended covering index was added to the set of candidates generated to see if Mindexer would

include it in its final recommendations. This did not occur, with two different runs showing that it ignores it and prefers its own candidates, as shown in Table 5.3. The candidate selection process was then turned off, and Mindexer Modified v2 was forced to rank three indexes: two from its previous output, as well as the covering index. Again, Mindexer ignored the covering index, and did not include it in its output ranking. These outputs are shown in Table 5.4.

| Run 1 | Run 2 |
|---|---|
| (_id, time, data) | (_id, time, data) |
| (_id, time, version) | (link_type, time, id1) |
| (_id, time, link_type) | |
| (_id, time, id1) | |
| (_id, time, visibility) | |
| ... | |

Table 5.3: Mindexer Modified v1's recommended indexes

| Run 1 | Run 2 |
|---|---|
| (_id, time, data) | (_id, time, data) |
| (link_type, time, id1) | (link_type, time, id1) |

Table 5.4: Mindexer Modified v2's recommended indexes

After the bug fixes were implemented in Mindexer v0.3.0, the covering index now consistently scored the highest and was chosen as the recommended index across all sample ratios. This indicated that the scoring system, while still with its flaws, was able to rightfully identify the developer-recommended index as the ideal candidate for LinkBench.

### Mindexer's Performance Tester Reliably Simulates Workload Performance

To get a more comprehensive understanding of why the four indexes were recommended by Mindexer v0.3.0 in Subsection 5.1.3, the performance tests were also carried out on Mindexer's built-in performance tester, as Mindexer only tests the non-modifying (fetch) queries, and the performance could be more equal across the recommendations. Similarly, the covering index was also tested on Mindexer for completeness.

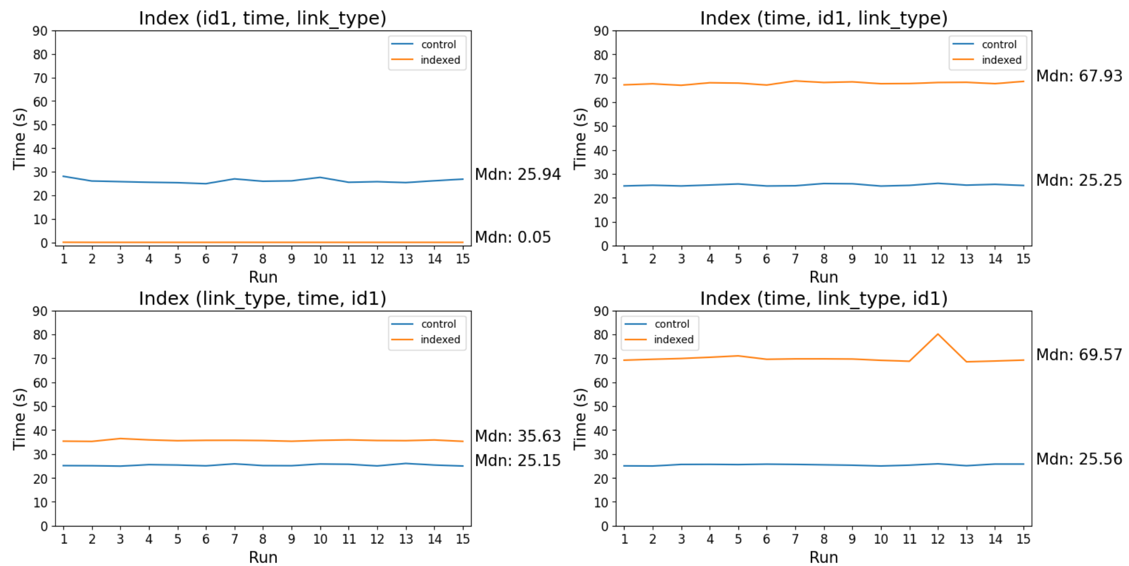



Figure 5.5: Mindexer v0.3.0 performance for each recommended index

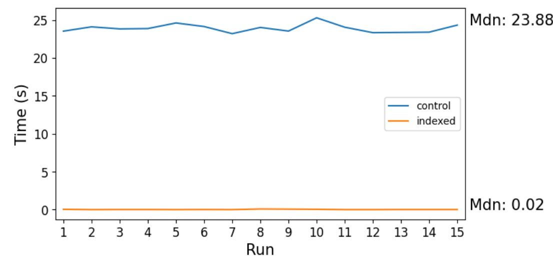



Figure 5.6: Mindexer v0.3.0 performance for the covering index

As observed in Figure 5.5, the indexes were also not performing identically, with the first index again being the only one that improved performance over the control. While it does not match the covering index (seen in Figure 5.6), it does come remarkably close. The two indexes with "time" as their first field maintain their notably bad performance, and should clearly have been excluded from Mindexer's recommended index output. These results do however show that Mindexer's built-in performance tester is able to reliably simulate the relative performance difference of its recommendations, and the significance of this is further discussed in Section 6.3.

# Chapter 6

# Analysis of Results

This chapter analyses the results identifying the weaknesses in the Mindexer recommender, and discusses insights into how the tests were carried out, notable reasons why the weaknesses exist, and suggestions for future work addressing these flaws.

## 6.1 Mindexer v0.2.0 Recommends Negatively Performing Indexes

In the initial evaluations, we identified that Mindexer v0.2.0 did not reliably suggest high-performing indexes, with many recommendations resulting in worsened performance. Alongside these issues, it also had large variations in its recommendations list, with the output producing 51 recommendations most of the time, but occasionally recommending only two or three indexes. The actual recommendations within each output set also differed, adding to the inconsistency of its results.

As part of this study, many of the bugs that were fixed and committed as updates to the codebase resulted in an improved consistency in Mindexer's recommendations output, with it now only recommending one index each time it was run. These fixes decluttered the candidate set that Mindexer was scoring, as the inclusion of the "_id" field resulted in the algorithm thinking that those indexes would have a much larger impact on performance, but this was not the case as the "_id" index already exists by default within MongoDB collections. There was also an issue within the recommendations algorithm that would prevent it from properly checking the recalculated score of subsequent indexes, and it did not catch that the index score was actually 0 or less. which was also fixed. Hence, many of these irrelevant indexes are no longer in consideration, reducing the likelihood of the top recommended index performing negatively. It should be noted that due to other design choices, there can still be recommendations that perform badly, but this can no longer be a result of the issues described in this section.

## 6.2 Imprecision in Mindexer's Cardinality Estimates

When testing the different scores Mindexer Modified v3 gives to the covering index depending on the sample ratio, we found that Mindexer's scoring system is flawed when it comes to assessing an index's impact on a query that returns very few results. This is especially obvious with LinkBench, with the majority of its queries (>95%) returning less than 10 values, as most of these are simple fetches on the "_id" field. Therefore, when Mindexer samples the database, it is very unlikely that the sample will contain matching data, regardless of the index being assessed, which effectively makes the query in question irrelevant, as it will not meaningfully affect the scores for any index under assessment. We observed that the majority of the differences in cardinality estimate scores when changing the sample ratio can be seen in three "outlier" queries, as the number of results returned by the control exceeds 200, crossing well into the 10000s with query #335. It is due to the existence of these complex queries that Mindexer is able to determine the impact of an index on the database, as it is able to more accurately produce a score estimate even with a small sample. From this, we can see that Mindexer's scoring is not equipped to handle scenarios where there is an abundance of simple queries due to an imprecision in its cardinality estimates, resulting in the scoring system relying too heavily on larger queries.

There are limitations to this test, as it specifically focused on the scores given to the covering index identified by LinkBench. For more conclusive results, the test could be repeated across various candidate indexes, and we can see if the observation of the sampling estimator relying on complex queries holds. Also, Mindexer automatically assigns a score of 0 for queries that the index is verified to not be usable on, which means that depending on the index being tested, there will be differences as to which queries are included for scoring. For reference, when testing the covering index, only 142 of the 387 queries were actually scored. This suggestion is explored in further detail in Section 8.1.

Changes that could help improve this flaw include widening the range of queries that Mindexer inspects, as it specifically narrows down its scoring against find queries only. In the LinkBench workload, there are many queries that find records and then modify them, and Mindexer could score indexes against the find portion of these queries. A comprehensive update could also take into account aggregation queries, but this is more challenging with various approaches available that vary in complexity. This suggestion is explored in further detail in Section 8.2.

## 6.3 Mindexer Variably Recommends Indexes of Differing Performance

In these performance tests, we first verified that Mindexer v0.3.0 was performing consistently when running on a sample ratio of 1, with scores remaining consistent across runs. Previously, the developer-recommended covering index was not identified as such due to a bug causing the algorithm to incorrectly determine that certain indexes could not be used for sorting, and did not grant it a “sort bonus”. Once this was fixed, the algorithm was correctly recommending the covering index in first place, even when full candidate generation was turned back on. With this, we moved on to testing how good the scoring system is with Mindexer v0.3.0’s generated candidates, as its field-length restrictions prevent it from being able to discover the covering index. Here, an issue that was identified was that the ordering of the candidates set is non-deterministic, resulting in the final recommended index differing between runs. This is because there are four indexes tied for first place, and Mindexer simply recommends the first in the set. However, we did not want to just sort the set into a list, as there was concern that the indexes were not truly identical in performance impact, and a simple sort could result in Mindexer always recommending the badly performing one. As a result, we decided to perform the full set of performance tests on these top four indexes, and see how they compare when applied to the LinkBench workload.

The results from the performance tests are rather interesting, as they demonstrate various benefits to utilising Mindexer, while also highlighting flaws in its approach. Firstly, we observed that Mindexer is clearly not recommending indexes of equal performance, as one improves performance drastically, and the others make it worse. It is worth noting that these four indexes used the same three fields, just in a different order, which shows the importance the index field order has on the performance of the workload. This weakness in Mindexer’s scoring is due to it not currently taking into account the Equality, Sort, Range (ESR) Rule as specified by MongoDB. Here, MongoDB states that index fields should be ordered by placing all fields that are used as exact matches first, then fields used for sorting, and finally any remaining fields used for range matches [16]. The two indexes that have ‘time’ as their first field suffer performance-wise due to this, as ‘time’ is used for both sorting and range matches. An update to the scoring system that factors this in and modifies the index candidate scores accordingly could resolve this problem, allowing Mindexer to consistently recommend an index that performs well. This suggestion is explored in further detail in Section 8.3.

Additionally, another noteworthy observation from this test was that the first index, (id1, time, link_type), performed identically to the covering index when used on LinkBench. This is quite significant, as Mindexer has found a 3-field index that results in similar performance when compared to a covering (7-field) index for the workload, which is particularly beneficial as it results in faster database updates and reduced index storage usage. The reduced update times are likely the reason why the performance of this smaller index matches the covering index on LinkBench but not on Mindexer, as Mindexer only benchmarks fetch queries, hence the covering index can marginally outperform it due to field coverage. An avenue of further research in this area could be to increase the number of fields Mindexer iterates over for candidate indexes, as it is currently limited to just three fields. Increasing the field size may result in the discovery of an index that outperforms that of the covering index, but this will come at the expense of longer running times for Mindexer. This suggestion is explored in further detail in Section 8.4.

These tests also demonstrated that Mindexer's built-in performance tester was able to reliably compare the performance of the workload with and without the recommended index. Here, the relative performance differences between the two tests were rather close to that of the actual differences when run on LinkBench. Mindexer has lower running times as it only tests the fetch queries, allowing it to complete the test in around 25 seconds on average instead of 431 seconds on average on LinkBench. This is beneficial as it reduces the load placed on the system and is an overall more efficient process. The relative differences, while not identical, are able to reliably communicate to the tester as to whether the index has a positive or negative impact on the workload's performance.

# Chapter 7

# Conclusion

Indexes are an important piece of the database puzzle, as they can significantly improve performance, even when used against a powerful workload, but have to be selected carefully. The experimental Mindexer tool attempts to solve the complex problem of discovering an efficient index while utilising a small sample of the database, and this study evaluated the effectiveness of its results when tested on the LinkBench workload. We presented a well-tested approach to evaluating the recommender, where we defined a process for assessing the performance of the tool's index recommendations against the workload in a controlled environment. Then, we identified several weaknesses in the implementation of Mindexer's recommendations and the design of its scoring system. Here, we discovered it was recommending negatively performing indexes, stemming from a cluttered initial candidate set and an issue in its recalculation algorithm, and introduced fixes to address it. We found that its cardinality estimates were imprecise and relied on complex queries, indicating that its scoring system is not equipped to handle workload scenarios with many simple queries. Finally, we observed that it variably recommends indexes of differing performance, as three undesirable indexes are scored identically to an excellent one, showing that its scoring system is flawed. These findings are significant, as they definitively conclude that Mindexer does not reliably recommend well-performing indexes on the LinkBench workload. However, investigations and tests on the codebase have discovered areas that could be worked on, as well as limitations to the existing tests that could be addressed. These future works include performing additional tests to verify that the sampler relies on complex queries, including other query types in the scoring system, the introduction of an improved index scoring algorithm using the ESR rule, and expanding the number of fields candidate indexes can have. From this continued effort comes the possibility that Mindexer could eventually reliably and accurately recommend efficient indexes, making it an incredibly useful tool that addresses a meaningful problem.

# Chapter 8

# Future Work

This chapter discusses four avenues of future work to improve the behaviour and results of the Mindexer recommender. These vary in scope, with some works that should bring immediate improvements, and others that lead to larger research with less predictable outcomes.

## 8.1 Performing Additional Sampling Tests

A key limitation in our tests which identified that Mindexer relies on complex queries due to imprecise cardinality estimates was that the test was only performed on the covering index, which meant that we could not verify that this would remain the case across all queries, especially since certain queries are skipped by the estimator if the index could not be used on them. A list of other indexes to test could be pulled from the existing Mindexer recommendations output, as this will give some insights into how exactly Mindexer arrived at the scores for those indexes. To test these, the code will have to be modified to utilise 10 or more sampling estimators, which was done by creating a list of estimators, and looping over them for each query the index is scored against. Alternatively, the Mindexer tool can be run multiple times, but that is a slower solution. Once the cardinality estimate and final benefit scores are calculated, the values can be printed on the screen for each sampling estimator in use. This test should then be repeated for the larger sample ratios, as the default is 0.001. A separate script can then be used to extract these values and graph them as necessary. The estimate score is the number of results Mindexer predicts the query will return based on the sample database it has, so this should be compared to a run performed on a sample ratio of 1, as that is the true value. If Mindexer is only able to reliably predict the score for queries with large results, then it relies on those complex queries. Another point to note here is the benefit score, which was not evaluated in these tests, so it may be a value of interest as it uses its own formula that we did not inspect further.

## 8.2 Including Modifying Queries

The Mindexer scoring algorithm currently scores indexes against queries that only find data, and therefore excludes other queries that could benefit from indexes, such as find and modify queries, or certain aggregations. A modification to the code to support this can be quite useful for the tool, as it would have more data points to score indexes against, and can therefore give a more informed recommendation that is likely to be useful for those queries. The first part of this suggestion (find and modify queries) is simpler to implement, and can be done by updating the query log filter used by Mindexer to also include find and modify queries, and adapting the code such that it pulls the necessary data from the find portion of the query while retaining prior functionality. For aggregations, it is a little more complex, but mainly due to how MongoDB handles indexes within the aggregation pipeline. Here, there are only specific situations where an index is to be used, and hence Mindexer should follow suit and only check those scenarios. A more holistic approach could include query optimisation to maximise index usage within the aggregation pipeline, but this is rather out of the scope of the tool. For MongoDB, standard indexes can only be utilised in early aggregation stages in one of three scenarios: (1) a match stage which is the first stage in the pipeline, (2) a sort stage which is used without a preceding project, unwind, or group stage, and (3) a group stage which is used using fields that were sorted on previously, and has an accumulator operator [17]. In scenario 1, Mindexer could check if the match stage is first, and include the fields within it as considerations for index candidates. The results returned from this stage can also be used within the index scoring algorithm to add to the estimate score. For the second scenario, Mindexer can be updated to check if a sort stage exists without the aforementioned prior stages, and if so, the sort fields could be used to check if a sorting bonus is available for each scored index. The third scenario is the most complex, as Mindexer is not currently configured in any way to meaningfully assess an index based on a group stage. Here, further investigations will have to be performed to see if the existing query logs have data that could be beneficial to scoring candidate indexes against the group stage in the pipeline.

## 8.3 Implementing the Equality, Sort, Range Rule

An identified weakness in the design of Mindexer's scoring system is that it does not factor in the Equality, Sort, Range (ESR) Rule as identified by MongoDB, which can be used to properly rank indexes that contain the same set of fields. Currently, Mindexer treats these indexes that share the same set of fields as if they are equal, but due to how indexes are actually utilised within queries, the ordering of these fields in the index turns out to be quite important. MongoDB states that index fields should be ordered by placing all fields that are used as exact matches first (in any order), then fields used for sorting (provided that all equality fields are utilised first), and finally any remaining fields used for range matches (provided that all sort fields are utilised first in-order) [16]. As each query performed by the given workload could have different usages for the fields, Mindexer should have its index scoring system updated such that the ESR rule is factored in on a query-by-query basis. This can be done by first splitting the query into three identified parts: (1) exact matches, (2) sorts, and (3) range matches. Here, if the assessed index follows the ESR rule for that specific query, it will get a bonus, similar to the sorting bonus given currently. Introducing this additional consideration within the scoring system will ensure that when the scores are totalled for each index candidate, the index with the best overall ESR compliance is more likely to be chosen. Ideally, the index from this updated system strikes a good balance between the various scoring factors, and demonstrates this balance in impressive real-world performance. However, this is something that can only be confirmed with an updated set of tests, carried out in a similar manner to that of our finalised testing process as identified in our first contribution, outlined in Section 4.1.

## 8.4 Expanding the Maximum Field Size

Mindexer currently limits itself to generating candidate indexes with up to three fields, with permutations generated based on the fields it finds are used by queries in the provided workload. As a result, many potentially good indexes, including the covering index, are not considered as Mindexer simply does not generate them as candidates. This is, however, an imposed limitation, as it is very easy to update the global variable to a higher value, allowing the code to generate larger index permutations. The complications that arise from increasing this limit are instead seen in the scoring system, as a larger set of candidates to score will result in longer running times for the tool. As seen in Table 8.1, the number of permutations is more than quadruple in size when the limit is increased to four fields, and the set size quickly gets out of hand with further increases to the limit.

| **Max Field Size** | **Candidate Set Size** | |
|---|---|---|
| | **Mindexer v0.2.0** | **Mindexer v0.3.0** |
| 3 | 400 | 259 |
| 4 | 2080 | 1099 |
| 5 | 8800 | 3619 |
| 6 | 28960 | 8659 |
| 7 | 69280 | 13699 |

Table 8.1: Mindexer's index candidate set sizes

It is, therefore, necessary that other improvements to the candidate generation system should be made if an increase to the field size is considered, such as a filter that removes candidates that are unlikely to score well prior to the tool actually scoring them. A potential implementation of a filter such as this could remove candidates if they do not abide by the ESR rule as described in Section 8.3. A more advanced implementation could utilise the Index Filter in the DISTILL prototype, which uses a pre-trained model [15]. These are merely suggestions, and their efficacy will have to be determined using updated tests, such as our finalised testing process defined in Section 4.1.

# Bibliography

[1] T. Rückstieß, “Mindexer.” https://github.com/mongodb-labs/mindexer, 2022. Online; accessed 5-March-2023.

[2] D. Borthakur, T. Armstrong, M. Callaghan, and T. Luo, “LinkBench.” https://github.sydney.edu.au/USYD-DBRG/LinkBenchCode, 2022. Online; accessed 13-March-2023.

[3] bsonspec.org, “BSON (Binary JSON) Serialization.” https://bsonspec.org/, 2023. Online; accessed 28-October-2023.

[4] IBM, “What is MongoDB?.” https://www.ibm.com/topics/mongodb, 2023. Online; accessed 18-October-2023.

[5] MongoDB, “Database Profiler.” https://www.mongodb.com/docs/manual/tutorial/manage-the-database-profiler/, 2023. Online; accessed 28-October-2023.

[6] Y. Han, Z. Wu, P. Wu, R. Zhu, J. Yang, L. W. Tan, K. Zeng, G. Cong, Y. Qin, A. Pfadler, *et al.*, “Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation,” *arXiv preprint arXiv:2109.05877*, 2021.

[7] C. Cioloca, M. Georgescu, *et al.*, “Increasing Database Performance using Indexes,” *Database Systems Journal*, vol. 2, no. 2, pp. 13–22, 2011.

[8] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman, “Index Selection for OLAP,” in *Proceedings 13th International Conference on Data Engineering*, pp. 208–219, IEEE, 1997.

[9] M. R. Frank, E. R. Omiecinski, and S. B. Navathe, “Adaptive and Automated Index Selection in RDBMS,” in *Advances in Database Technology—EDBT’92: 3rd International Conference on Extending Database Technology Vienna, Austria, March 23–27, 1992 Proceedings 3*, pp. 277–292, Springer, 1992.

[10] S. Choenni, H. Blanken, and T. Chang, “Index Selection in Relational Databases,” in *Proceedings of ICCI’93: 5th International Conference on Computing and Information*, pp. 491–496, IEEE, 1993.

[11] S. Chaudhuri and V. R. Narasayya, “An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL server,” in *VLDB*, vol. 97, pp. 146–155, San Francisco, 1997.

[12] S. Chaudhuri, M. Datar, and V. Narasayya, “Index Selection for Databases: A Hardness Study and a Principled Heuristic Solution,” *IEEE transactions on knowledge and data engineering*, vol. 16, no. 11, pp. 1313–1323, 2004.

[13] J. Kossmann, S. Halfpap, M. Jankrift, and R. Schlosser, "Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2382–2395, 2020.

[14] J. Kossmann, A. Kastius, and R. Schlosser, "SWIRL: Selection of Workload-aware Indexes using Reinforcement Learning.," in *EDBT*, pp. 2–155, 2022.

[15] T. Siddiqui, W. Wu, V. Narasayya, and S. Chaudhuri, "DISTILL: Low-Overhead Data-Driven Techniques for Filtering and Costing Indexes for Scalable Index Tuning," *Proceedings of the VLDB Endowment*, vol. 15, no. 10, pp. 2019–2031, 2022.

[16] MongoDB, "The ESR (Equality, Sort, Range) Rule." https://www.mongodb.com/docs/manual/tutorial/equality-sort-range-rule/, 2023. Online; accessed 16-October-2023.

[17] MongoDB, "Aggregation Pipeline Optimization." https://www.mongodb.com/docs/manual/core/aggregation-pipeline-optimization/, 2023. Online; accessed 24-October-2023.