UNIVERSITÄT
DES
SAARLANDES

**Faculty of Mathematics and Computer Science**
**Department of Computer Science**

# Workload-based Data Partitioning for Index Construction

**Bachelor's Thesis**

written by

**Lukas Wilde**

**31st August 2022**

Supervisors
**Jens Dittrich**

Advisor
**First Advisor**

1st Reviewer
**Jens Dittrich**

2nd Reviewer
**Second Reviewer**

## Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

## Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

## Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

## Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) acessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, _____ _____

Datum/Date Unterschrift/Signature

# Acknowledgement

# Abstract

# Contents

**1 Introduction** **1**

**2 Related Work** **2**

**3 Background** **7**
3.1 Partitions (???and Partitioning functions) . . . . . 7
3.2 Hybrid Index Structures . . . . . 8
3.3 Numerical Differentiation . . . . . 8

**4 Framework** **9**
4.1 Overview . . . . . 9
4.2 Partitioning algorithms . . . . . 10
4.2.1 Partitioning by Frequency . . . . . 11
4.2.2 Partitioning by Purity . . . . . 12
4.3 Index Modifications . . . . . 12
4.3.1 Changing leaf data structure . . . . . 12
4.3.2 Moving leaves higher up . . . . . 12
4.4 Workload Generation . . . . . 12

**5 Evaluation** **13**
5.1 Setup . . . . . 13
5.2 Datasets and Workloads . . . . . 13
5.3 Role of Partitioning Parameters . . . . . 13
5.4 Lookup Performance . . . . . 13
5.4.1 Frequency Algorithm . . . . . 13
5.4.2 Purity Algorithm . . . . . 13
5.4.3 Frequency Algorithm . . . . . 13
5.4.4 Purity Algorithm . . . . . 13

**6 Conclusion and Future Work** **15**

**A Appendix** **i**

# Chapter 1

# Introduction

Here is a citation [1].

- DBMS routinely use index structures for increased performance
- Index pre-configured or chosen by user
- Mostly no utilization of underlying data or workload distribution
- Except: learned indexes -> Related Work
- Motivation: different data structures for different query workloads (hash table?)
- For this, introduce concept of hybrid index structures
- Create partitions to create singular indexes and combine them
- Optimize partitions based on one/multiple metrics
- As motivation: GENE, starting point for generic search
- Introduce what is covered in what section of this thesis

# Chapter 2

# Related Work

In this Chapter, we first introduce the index structures that served as a baseline to compare our algorithms. We cover traditional tree-like index structures like the B$^+$-tree, look into Radix Trees represented by the Adaptive Radix Tree and then proceed to Learned index structures like the Recursive Model Index and the Piecewise Geometric Index. After that, we explore two fields in which the query workload is used in data partitioning: Adaptive Hybrid Indexes and Distributed Database Systems.

When we look at traditional tree-like index structures, the well-known B$^+$-tree [2] is the first index that comes to mind. It was designed to index a dynamically changing file with the assumption that only a small part of the index can remain in main memory at all times. The authors acknowledged that this file can be subject to change, and as such one would need efficient ways to not only search the index, but also enable the insertion and deletion of existing keys. The nodes in a B$^+$-tree are always containing at least $k$ and at most $2k$ keys at a time, resulting in a space occupancy of at least 50%. The keys in inner nodes act as boundaries to guide the retrieval, essentially producing disjoint ranges. Should a key fall into a specific range, the corresponding child pointer is used to get to the node of the next layer. The keys in the nodes are ordered, which allows for an efficient binary search. Insertion and deletion follow the same principle, except when inserting or deleting a key would result in a key count of less than $k$ or more than $2k$ inside a node. In this case, neighboring nodes need to be merged or a node needs to be split in order to guarantee the invariant that each node contains between $k$ and $2k$ keys. The advantage of this sort of index is that the keys are kept ordered, which enables efficient predecessor/successor queries and range queries. However, B$^+$-trees have a poor cache utilization, since half the space is needed for the child pointers. There are several variants of the B$^+$-tree that try to improve the caching behavior, e.g. the Cache Sensitive B$^+$-tree (CSB$^+$-tree) [3]. Their main idea is that only one child

pointer is stored explicitly and every other children can be found by adding a specific offset to that first address. This requires that child nodes are stored contiguously in memory, resulting in a overhead when nodes need to be split or merged.

The next family of indexes is Radix Trees, with the Adaptive Radix Tree (ART) [4] as a representative. The authors recognize that the $B^+$-tree is widely used for disk-based database systems but believe it is unsuitable for main-memory databases. This is mainly due to traditional index structures' inefficiency in CPU caching. Additionally, there is the problem of stalls in the modern-day CPU pipeline, which is caused by the CPU's inability to easily predict the result of comparisons. As comparisons are necessary to traverse a $B^+$-tree, this causes more latency for the index. To overcome these problems, the authors introduce an improvement to Radix Trees, which uses certain parts of the keys directly to guide the search in the tree. While Radix Trees get rid of the previously mentioned CPU stalls, they often have to make a global trade-off between tree height and space efficiency. This problem is solved by introducing adaptive nodes with varying capacities to hold child pointers. Results indicate that ART can outperform other main-memory data structures, with the only competitor being a hash table. However, as these store keys in a random order, they cannot support range queries efficiently and are only useful in specific scenarios.

The last family of indexes are Learned index structures, a type of index that only emerged recently. Learned index structures generally try to leverage recent progress in the field of Machine Learning to improve index performance. The Recursive Model Index (RMI) [5] introduces the concept that indexes are models that simply map keys to positions in a sorted array. The authors state that most modern index structures do not consider the data distribution and miss out on highly relevant optimizations. While most datasets don't follow simple patterns, they argue that Machine Learning (ML) approaches can be used to incorporate these patterns. One can look at finding the position of a key by traversing a $B^+$-tree as slowly reducing the error. While at the start, one would need to search in every possible location, after the first comparison in a $B^+$-tree, there is only a subset of locations left (i.e. the right or left sub-tree of the root). The same mentality can be adapted to ML models with a slight difference: instead of needing to be certain that a specific key is for example located in the left sub-tree, it would be enough for the key to be in the left sub-tree with a high probability. The authors argue that while it is hard to guarantee that a single ML model will reliably reduce the error from millions of possible locations to hundreds for the final search, it is reasonable for a model to do so from millions to tens of thousands. As these tens of thousands of locations are too large to search for the final position of the key directly, they construct a hierarchy of ML models, where each model picks the next layer's model that should

be used to predict the position of the key. This hierarchy does not need to follow a strict tree structure; each model can cover an individual number of keys. The benefit of this architecture lies in the ability to customize the models. For example, the bottom layer of nodes could only represent linear regression models (as there are many leaves and linear regression models are quite inexpensive), while higher up, one could theoretically use more complex neural network structures. In practice, however, neural networks are quite time-consuming to evaluate, which is why mostly (linear) models are used. The data segmentation happens through the structure and training of the internal node's models. While this paper introduces the use of the underlying data distribution for index construction, it suffers from the explainability of neural networks. Therefore, no clear properties could be used for my work.

FITing-Tree [6] tries to combine the flexibility of traditional index structures with learning by indexing linear data segments. The authors argue that ML models can not only be used to speed up the lookup performance of index structures, but it is also possible to reduce the memory requirements of indexes. Recent results [7] have shown that index structures in OLTP databases can take up to 55% of the available memory, making it all the more enticing to develop indexes that perform similarly to the state-of-the-art while reducing memory consumption. The data partitioning is done by a single pass over the sorted data. The segmentation algorithm aims to determine the data segments' bounds so that the relation between keys and positions in the sorted array can be approximated by a linear function. To give reliable performance estimates, an error parameter is used to indicate how much an estimated position is allowed to deviate from the real position. A new segment is created once a point falls outside an error cone that ensures this maximum deviation. Otherwise, the cone is adjusted by tightening its bounds. Once the segments are determined, they are indexed by a $B^+$-tree to find a key's corresponding segment, and a binary search is performed inside the segment to find the actual position of the key.

The authors of the Piecewise Geometric Model index (PGM) [8], which we used as the third baseline in the evaluation, tried to improve upon the ideas of FITing-Tree. While FITing-Tree's approach seemed reasonable, a disadvantage was the data segmentation. The authors note that the single-pass segmentation algorithm does not produce the optimal number of data segments, leading to more data segments, a larger tree height, and increased lookup time. By reducing the segmentation to the problem of constructing a convex hull and allowing the index to be built recursively, they could increase the lookup time and ensure provably efficient time and space bounds in the worst case.

TODO: Distribution-aware PGM

While learned index structures perform so well because they can adapt

to the underlying data distribution, apart from the distribution-aware PGM, they do not consider the workload that will be executed. RMI partition the data indirectly through their models, FITing-Tree and PGM explicitly use segmentation algorithms before building the index to determine the data that belongs in one segment. However, workload information might be beneficial to index construction, e.g. by indicating that certain data segments are not frequently requested. My work covers whether workload information can be used to improve data segmentation and thereby yield better performance.

Adaptive Hybrid Indexes [9] tackle the problem of selecting suitable encodings inside index structures to trade-off between space utilization and index performance. Compact indexes reduce the index's memory, allowing the database system to utilize that free memory to accelerate queries. This is achieved by either being able to keep a larger working set in memory or, when there's a memory budget for indexes, by enabling the use of more index structures that are kept in main memory. However, they are naturally inferior to performance-optimized indexes. The decision of what encoding should be used on which part of the index is hard to make at build-time. Therefore, the authors propose to make these decisions at run-time. They introduce a framework that allows for monitoring the accesses across the index nodes when queries are processed, and based on these metrics, they classify whether nodes are cold or hot. Using so-called context-sensitive heuristic functions (CSHF), the framework determines, based on the classification of hot- and coldness, the memory budget, the historical classifications, and other properties, whether a node should have a compressed or performance-optimized encoding. Especially relevant to my work is the classification as hot or cold. Once the data is split into segments and inserted into a tree-like structure, it could be beneficial to modify the index based on this classification. While the authors do this at run-time, my work focuses on analyzing the workload before building the index. They either use performance or memory-optimized encodings of nodes to represent hot or cold data, but one could also consider shifting leaves in the tree higher up to optimize for cache benefits.

Distributed database systems are another field where the workload is used for partitioning. An example there is Schism [10]. The motivation behind this approach is to improve the performance and scalability of distributed databases. Each tuple is represented as a node in a graph. Two nodes are connected if the corresponding tuples occur in the same transaction. The edges are weighted with the total amount of co-occurrences in transactions. Given a number of partitions $k$, the algorithm will find a set of cuts of the edges that produces $k$ distinct partitions with roughly equal weight and minimal costs along the cut edges. The intuition behind this is that tuples

that are often accessed in the same transaction should also reside in the same partition/node to optimize query processing. By minimizing the cost along the cut edges, pairs of tuples that are seldom accessed together are split into different partitions, whereas often connected tuples stay in the same partition. While we do not look at transaction-based workloads in this work, there is a similarity in looking at workload properties to partition the data. Schism uses the frequency of co-occurrences to do this partitioning, which indicates that the frequency of query accesses could be a promising property to look at

# Chapter 3

# Background

In this chapter, we cover the background information necessary to understand our approach and the idea behind the used algorithms.

First, we look at the definition of partitions and how indexes partition data using partitioning functions in section 3.1. After that, we cover hybrid indexes and their structure in section 3.2. To understand the idea behind an algorithm that will be used, we look at numerical differentiation to approximate the derivative of a function in section 3.3.

## 3.1 Partitions (???and Partitioning functions)

Let us first look at the definition of a partition in the rigorous mathematical sense to transfer this to the field of index structures. The following definition is taken from Lucas [11].

**Definition 3.1 (Partition):**
Let $M \neq \emptyset$ be a nonempty set. A partition $P$ of $M$ is a collection of subsets of $M$ with the following properties:

P.1 $\forall p \in P : p \neq \emptyset$

P.2 $\forall p, q \in P : p \neq q \implies p \cap q = \emptyset$

P.3 $\bigcup_{p \in P} p = M$

To summarize, a partition $P$ of $M$ is a collection of nonempty (P.1), mutually disjoint (P.2) subsets of $M$, whose union exhausts all of $M$ (P.3).

To adapt this to data partitioning for index construction, we can look at the keys $K = \{k_1, k_2, ..., k_n\}$ over which we want to construct our index. If we look at typical $B^+$-trees, the data partitioning is induced by the contents of the leaf nodes. $B^+$-trees don't allow empty nodes (P.1), there is only one

possible way to traverse through the index given a certain key, which means that the leaf nodes' contents are disjoint (P.2) and if the index was built over the whole key space $K$, every key will be present in some leaf node (P.3).

- Partitioning functions for indexes
- Used in routing of through index

## 3.2 Hybrid Index Structures

TODO: Need this to describe structure of our index TODO: read into different kinds of hybrid indexes

- What are hybrid index structures?
- Advantages: optimize for subproblems, combine to one index
- challenges: correct combination of these structures (e.g. routing through data structure)

## 3.3 Numerical Differentiation

TODO: visualization (Sekanten) To calculate the derivative of a function $f$ at a point $x$, we know we have to evaluate the limit

$$\lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

However, we cannot evaluate this limit if we only have a discrete function. For this purpose, we can use finite differences to approximate the derivative. There are three common ways for this kind of approximation:

1. Forward difference approximation: $\frac{f(x+h)-f(x)}{h}$
2. Backward difference approximation: $\frac{f(x)-f(x-h)}{h}$
3. Central difference approximation: $\frac{f(x+\frac{h}{2})-f(x-\frac{h}{2})}{h}$

As these are only approximations, they are not exact representations of the actual derivative and produce an error. Using Taylor's expansion, we can show that for the central difference approximation, this error is in $O(h^2)$ while it is in $O(h)$ for the forward and backward difference approximations. The central difference approximation is, therefore, more accurate but has the downside that it can yield zero estimations for periodic functions.

# Chapter 4

# Framework

This chapter introduces the framework that was implemented to generate workloads, partition data and eventually benchmark a custom index that uses the partitioning.

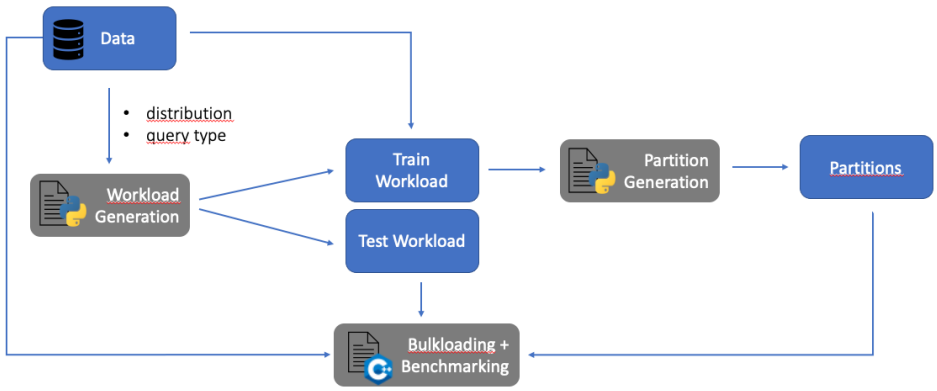


Figure 4.1: Framework Overview

## 4.1 Overview

As we can see in Figure 4.1, the origin of all processes is the underlying data that should be indexed. Using a python script, we can specify properties like the distribution and type of queries (point, range) that should be generated to access the data through the index. This is done by specifying a series of `Partition` objects that wrap the following information:

- `qtype`: The query type that should be generated in this section
- `num`: The number of queries for this section
- `distribution`: Distribution underlying the generated queries
- `index`: Whether the generation happens index-based or domain-based
- `min, max`: minimal and maximal values that indicate the section boundaries. Either index or domain-based, depending on the value of `index`.

This gives us a very flexible way to generate arbitrary workloads. Note that while a partition generated for the index construction later satisfies that the elements are mutually disjoint, the `Partition` objects used for workload generation do not need to be disjoint. This only means that we can overlap the boundaries of the objects to generate even more flexible workloads. In fact, it is the only way to generate regions of the data that are accessed through multiple types of queries, e.g. through both point and range queries. The queries generated by this step are divided into a train and test workload and saved to files for later use in the C++ benchmarking.

TODO: Example of overlapping distributions

Given the train workload, the partitioning algorithms that are described in Sections 4.2.1 and 4.2.2 can analyze the corresponding properties of the workload and will produce a partition of the underlying data. The resulting elements are saved to a file with additional information that can be used for the index construction. With this partition, the index is bulkloaded from the data where each element of the partition corresponds to an individual leaf node. The additional information like relative frequency and predominant query type of a segment can be used to modify the index. For example, if only point queries access a segment, it could be beneficial to manage data access through a hash table instead of a normal B-tree leaf. The next step is to execute the test workload on the index and compare it to other state-of-the-art indexes that were introduced in Section 2, namely a $B^+$-tree, an Adaptive Radix Tree (ART) and a Piecewise Geometric Model index (PGM).

## 4.2 Partitioning algorithms

There are a plethora of properties that one could look at when analyzing query workloads, but inspired by the works in Section 2, we decided to focus on two properties and look at how we could use these to partition the data and create segments. As described in the previous Section, we can use the train workload to partition the data. The test workload is immediately saved to file after creation and not seen by both partitioning algorithms.

### 4.2.1 Partitioning by Frequency

The first algorithm analyzes the frequency of query access for each key in the key space. The motivation behind using the frequency as partition property is that hopefully we can benefit from caching effects during execution of the test workload. We would hope that highly frequent segments remain in cache so that subsequent queries can retrieve the location of the corresponding keys faster. Additionally, by analyzing the frequency, we can use that information to change the structure of the index. A first idea in this regard would be to shift highly frequent segments higher up in the tree, to prevent expensive pointer chasing when traversing the index.

We first realized, that key-by-key comparisons are not useful for a generalized partitioning algorithm because we can only operate on a train workload that is sampled from the general workload distribution. If we would use these key-by-key comparisons for the frequency to create partitions, we would probably overfit to the patterns in the train workload, even though these might only be caused by noise and not be present in the general distribution. Therefore, we employ an approach that tries to maximize the previously mentioned goal: find partitions where keys are accessed roughly the same amount of times. This partition should create elements, that utilize caching. Regions with almost no accesses will be put in one partition which will result in the segment being not loaded very often. On the other hand, regions with similarly frequent keys will result in the the corresponding segment remaining in cache if the frequency is high enough.

We use a single-pass algorithm that tries to find plateaus in the workload distribution by calculating the average change in frequency over a sliding window. It uses three phases depending on where it currently is with respect to a plateau, which are heavily inspired by the finite difference approximations from Section 3.3:

1. Start calculating a discrete forward difference approximation. As only keys "in the future" are considered, this phase is predestined to find an incoming plateau by checking if the calculated slope is near 0.

2. After such a plateau was found, we use the central difference approximation to establish the boundaries of the plateau. We use this approximation now, as it considers keys from before and after the current one. This should give a better estimation of when a plateau is ending.

3. Once the central approximation indicates that a plateau is ending, we switch to calculating the backward finite difference approximation to ensure that we find the exact end point of the plateau. We only consider previous keys, as we now know that an end is coming and this gives us the best chance to catch the key that is responsible for significantly changing the slope.

TODO: put algorithm code here

### 4.2.2 Partitioning by Purity

- Motivation: optimize index for different query types
- Algorithm

## 4.3 Index Modifications

### 4.3.1 Changing leaf data structure

### 4.3.2 Moving leaves higher up

## 4.4 Workload Generation

# Chapter 5

# Evaluation

This chapter will deal with the evaluation of the experiments

## 5.1 Setup

- hardware
- index parameters like slot size, PGM epsilon etc.

## 5.2 Datasets and Workloads

## 5.3 Role of Partitioning Parameters

- $$window_size$$
  , delta for frequency
- $$window_size$$
  for purity (as of yet)

## 5.4 Lookup Performance

### 5.4.1 Frequency Algorithm

### 5.4.2 Purity Algorithm

### 5.4.3 Frequency Algorithm

### 5.4.4 Purity Algorithm

# Chapter 6

# Conclusion and Future Work

- Previous results reproducable?
- What have we found?
- Does partitioning yield better lookup times?
- Is it beneficial to move leaves higher up in tree?
- Is it beneficial to use hybrid index structures (i.e. change layout/data structure in nodes)
- Best case/worst case considerations?
- Future Work: Combination of metrics
- Future Work: Look at more data structures other than BinarySearch-Leaves and Hashtables
- Future Work: What other workload metrics can be used for partitioning?

# Bibliography


[1] Douglas Comer. 'Ubiquitous B-Tree'. In: *ACM Comput. Surv.* 11.2 (June 1979), pp. 121–137. ISSN: 0360-0300. DOI: 10.1145/356770.356776. URL: https://doi.org/10.1145/356770.356776.

[2] R Bayer and E McCreight. 'Organization and maintenance of large ordered indices'. In: *Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control - SIGFIDET '70*. Houston, Texas: ACM Press, 1970.

[3] Jun Rao and Kenneth A. Ross. 'Making B+- Trees Cache Conscious in Main Memory'. In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. SIGMOD '00. Dallas, Texas, USA: Association for Computing Machinery, 2000, pp. 475–486. ISBN: 1581132174. DOI: 10.1145/342009.335449. URL: https://doi.org/10.1145/342009.335449.

[4] Viktor Leis, Alfons Kemper and Thomas Neumann. 'The adaptive radix tree: ARTful indexing for main-memory databases'. In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 2013, pp. 38–49. DOI: 10.1109/ICDE.2013.6544812.

[5] Tim Kraska et al. *The Case for Learned Index Structures*. 2017. DOI: 10.48550/ARXIV.1712.01208. URL: https://arxiv.org/abs/1712.01208.

[6] Alex Galakatos et al. 'FITing-Tree'. In: *Proceedings of the 2019 International Conference on Management of Data*. ACM, June 2019. DOI: 10.1145/3299869.3319860. URL: https://doi.org/10.1145/3299869.3319860.

[7] Huanchen Zhang et al. 'Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes'. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 1567–1581. ISBN: 9781450335317. DOI: 10.1145/2882903.2915222. URL: https://doi.org/10.1145/2882903.2915222.

[8] Paolo Ferragina and Giorgio Vinciguerra. 'The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds'. In: *PVLDB* 13.8 (2020), pp. 1162–1175. ISSN: 2150-8097. DOI: 10.14778/3389133.3389135. URL: https://pgm.di.unipi.it.

[9] Christoph Anneser et al. 'Adaptive Hybrid Indexes'. In: *Proceedings of the 2022 International Conference on Management of Data*. ACM, June 2022. DOI: 10.1145/3514221.3526121. URL: https://doi.org/10.1145/3514221.3526121.

[10] Carlo Curino et al. 'Schism'. In: *Proceedings of the VLDB Endowment* 3.1-2 (Sept. 2010), pp. 48–57. DOI: 10.14778/1920841.1920853. URL: https://doi.org/10.14778/1920841.1920853.

[11] John F. Lucas. *Introduction to Abstract Mathematics*. Lanham, Maryland: Rowman & Littlefield, 1990. ISBN: 978-0-912-67573-2.

[12] Jialin Ding et al. 'ALEX: An Updatable Adaptive Learned Index'. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, June 2020. DOI: 10.1145/3318464.3389711. URL: https://doi.org/10.1145/3318464.3389711.

[13] Jens Dittrich, Joris Nix and Christian Schön. 'The next 50 years in database indexing or'. In: *Proceedings of the VLDB Endowment* 15.3 (Nov. 2021), pp. 527–540. DOI: 10.14778/3494124.3494136. URL: https://doi.org/10.14778/3494124.3494136.

# Appendix A

# Appendix