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

Supervisor
Jens Dittrich

Advisors
Joris Nix
Christian Schön

1st Reviewer
Jens Dittrich

2nd Reviewer
Felix Schuhknecht

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

List of Figures

List of Tables

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

Before we can look at the approach that we used to answer the research questions, we need to cover some essential information. 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, where we also lay the foundation for the index that we used later on. 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).

To formalize the partitioning inside a (logical) index, we need to look at the (logical) nodes separately. According to Dittrich, Nix and Schön [12], one can define a logical node as follows:

Definition 3.2 (Logical Node): A logical node is a tuple (p, RI, DT):

1. $p : [R] \rightarrow D$ is a **partitioning function** on the relational schema $[R] : \{[A_1 : D_1, ..., A_n : D_n]\}$. p may be undefined.
2. $RI : D \rightarrow \mathcal{P}(N)$ is the **routing information**, where N is a set of nodes and $\mathcal{P}(N)$ is the power set of N. Each element of D (target domain of p) is mapped to a subset of nodes in N. RI may be undefined.
3. DT is the **data**. It is a set of tuples with relational schema $[R]$. DT may be empty.

Using this definition, we can see how a partitioning functions interacts with the routing information to organize an index. When we look at the example from above, we can see that the partitioning inside a B$^+$-tree happens through the partitioning function $p(t) = t.key$ and the routing information maps certain ranges (that are created in the B$^+$-tree through the keys in inner nodes) to the corresponding child nodes.

//TODO: grafik aus GENE papier

//TODO: noch ein Beispiel??

3.2 Hybrid Index Structures

To understand the motivation of this work and the general structure of the index that will later be used in the evaluation of partitioning algorithms, we will now look at the framework presented by Dittrich, Nix and Schön [12] in more detail.

We have already seen logical nodes in definition 3.2, which introduced the concept of partitioning function and routing information to construct an index. Although we have use the B$^+$-tree as an example above, we need to note that the routing information is more general than just a tree-like structure. In theory, there is no restriction to the layout of the graph that is created by the routing information. Additionally, this framework is purely logical yet, i.e. no physical layout is assumed. As long as all the nodes in the target domain of RI are part of the graph, we call this graph **Complete Logical Index**. The power of this indexing framework lies in its generalisation of common index structures. As we have seen, we can represent B$^+$-trees,

Radix Trees and Learned indexes trough this framework in the form of logical nodes. And as a complete logical index is just a graph of logical nodes, we easily can connect different node types to form a **Hybrid Logical Index**.

In the next step to realize this framework, we have to move from the purely logical index to a physical index by specifying how the logical node with its routing information and data should be physically represented. The first part to this specialization is to determine the search method that should be used in the RI and DT parts of the node to find key/value pairs. Presented options include linear search, binary search, interpolation search, exponential search, chained hashing and linear regression. Note that the choice here can already have an impact on the data layout that should be used inside the node, e.g. for binary search, we naturally require a sorted layout. Other options for the data layout include column vs row layout, whether to use a compression and which one or any other suitable hybrid data layout.

With this multitude of options to create a physical index, we need to decide which configuration we should use for a specific use case. This is done using a genetic breeding algorithm. It starts with an initial population of physical index structures and performs a set amount of iterations in the genetic search. This includes sampling from the current population and determining the fittest index together with the median fitness across the sample. After that, several mutations are randomly drawn for this fittest index and applied. If this new mutated index has a better fitness than the median fitness from before it is added to the population (replacing the worst index there if needed) and the whole process is repeated. After the set amount of iterations, the fittest index in the population is selected. The possible mutations are drawn from a distribution that allows to prioritize certain mutations over others. Mutations include changing the data layout or the search method of the RI or DT part, merging sibling nodes horizontally/vertically or splitting nodes horizontally/vertically. Note that the selection of the fittest index is completely dependent on the selected performance measure and the fitness functions representing it. A possible goal of the genetic optimization could be the runtime of the index on a given workload, where the fitness function could just be the median runtime of the workload over a given number of runs. Other goals include memory or energy-efficiency.

As the amount of iterations is set as a global parameter, the starting population plays a big role in how good a genetically bred index can become. If we would already start with a reasonable population, chances are higher that a good index can be found in the set amount of iterations. The authors present four possible ways to determine the initial population:

1. We start with a single physical node that does not contain data yet.

2. We start with a single physical node that contains all the data with a random or manually set data layout and search method.

3. We use bottom-up bulkloading where data layout and search method are picked randomly. The logical index structure is very similar to a B^+-tree in this case.

4. We start with a index that represents a state-of-the-art hand-crafted index. The genetic algorithm here serves mainly for fine-tuning this already established index.

Note that these starting options do not incorporate the workload information that will be used to find the fittest index in the population. However, if we were to already use that information in the starting population, we could be able to find a better starting point for the genetic search than the described options above. This is a main motivation behind my work: using the workload information to find a partitioning that can be used for the bulkloading of an index similar to point 3) above. It would be still fairly general and allow the genetic search a good amount of modifications, while setting physical design options optimized for the workload already in the beginning.

- 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: visulize disadvantage: periodic function

In the coming sections, we will discuss an algorithm that aims to find the boundaries of a plateau of a function. As we will not work on a continuous function, but a discrete one, we need to look at a way that enables us to approximate the derivative of a discrete function. This will be useful, as a plateau is identified by a vanishing derivative.

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: $\frac{f(x+\frac{h}{2})-f(x-\frac{h}{2})}{h}$

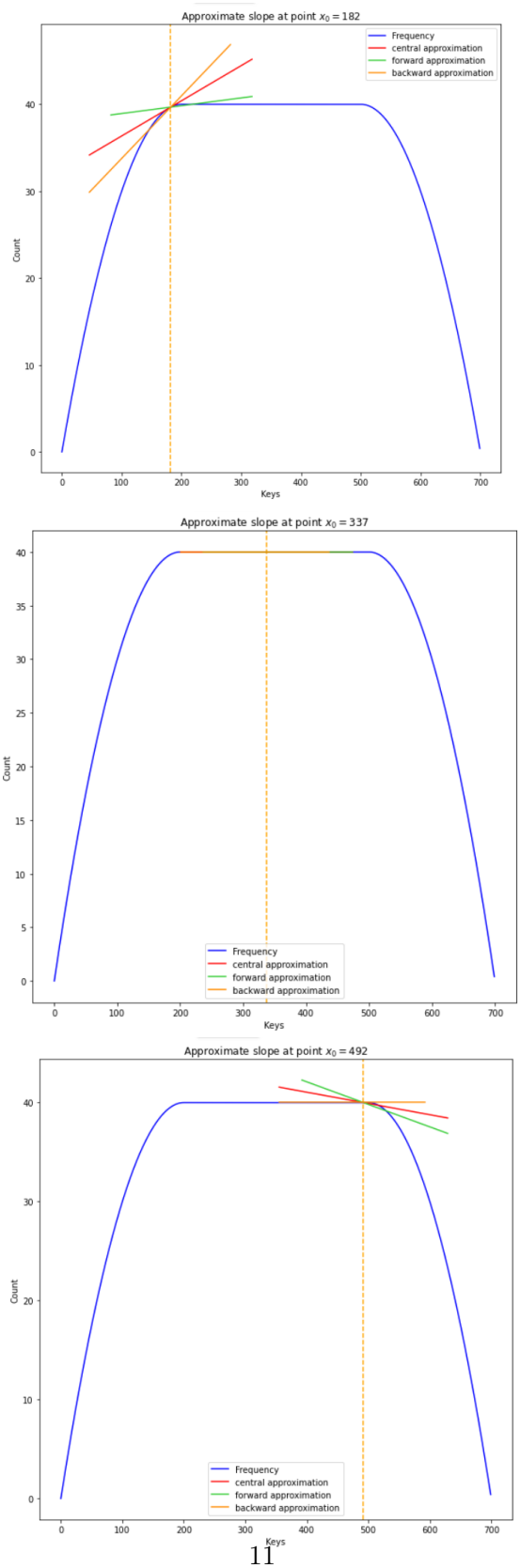

Figure 3.1: Comparison of the difference approximations

To motivate the later use of the approximations for identifying plateaus, we can look at Figure 3.1. All images show the three difference approximations at different points, where they are used to approximate the slope of the blue function. In the first image, we can see that we are at the start of the plateau, but only the slope of the forward approximation is nearly zero, the other two are still quite different to zero. This is why we can use the forward approximation to find the start of a plateau. On the second image, we see that when we are inside the plateau, all approximations are zero, so we would choose the central approximation as it is more robust to outliers in one single directions. And in the last image, we see the approximations still quite a bit before the end of the plateau, and we can recognize that only the backward approximation is still near zero. If we would use one of the others to find the end of a plateau, we had already stopped before the actual end is reached.

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. Note that the terms partitions and segments will be used interchangeably in the following.

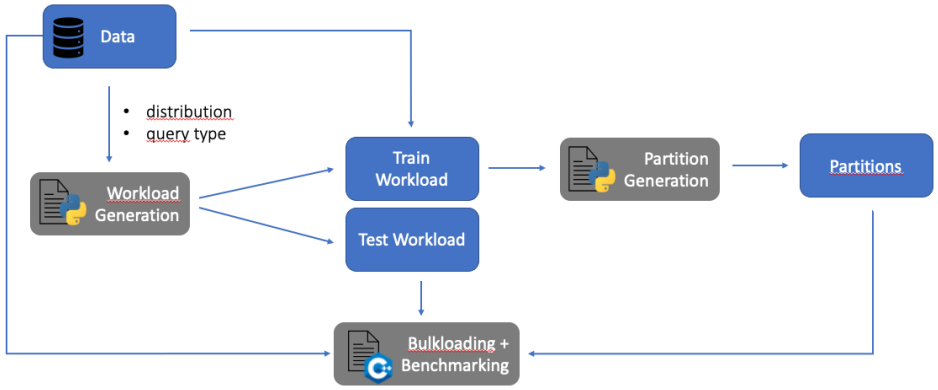

Figure 4.1: Framework Overview

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. 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 4.3.1 and 4.3.2 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 Workload Generation

The ultimate goal of this work is to generate good partitions for the index construction like it was mentioned in Section 3.2. The inputs to the partitioning algorithms are therefore a dataset and a workload sample which we know is representative of the expected workload. However, for the purpose of evaluating and testing the algorithms and modifications later used, we were in need of a flexible way to generate workload data, especially since it proved hard to find available real-world workload data. The workload generation is done in a Python script specifying a series of `Region` objects that wrap the following information:

- `qtype`: The query type that should be generated in this section, e.g. point or range queries
- `num`: The number of queries for this section
- `distribution`: Distribution underlying the generated queries, e.g. normal, uniform, ...
- `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 `Region` 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.

4.3 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.

4.3.1 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.

Algorithm 1 Partition by Frequency

$idx \leftarrow 0$
$n \leftarrow data.size$
while $idx < n - w$ **do**
  $current\_freq \leftarrow freq[idx]$
  $fwd\_mean \leftarrow mean(freq[idx : idx + w])$
  $fwd\_slope \leftarrow fwd\_mean/w$
  if $isclose(fwd\_slope, 0, delta/w)$ **then**
    $potential\_start \leftarrow idx$
    $idx \leftarrow idx + 1$
    for i in $1..w//2$ **do**
      $central\_left \leftarrow mean(freq[idx - i : idx + 1])$
      $central\_right \leftarrow mean(freq[idx + 1 : idx + w//2 + 1])$
      $central\_slope \leftarrow (central\_right - central\_left)/(w/(2 + i + 1))$
      if $!isclose(central\_slope, 0, delta/w)$ **then**
        $idx \leftarrow potential\_start$
        break
      end if
    end for
    if $idx == potential\_start$ **then**
      $idx \leftarrow idx + 1$
      continue
    end if
  end if
end while

//TODO: complete algorithm code here

4.3.2 Partitioning by Purity

The second algorithm does not analyze the frequency of a key, but the query types that access the key. This way, we can distinguish between keys that are not requested at all by queries, those that are only accessed by one single type (e.g. point or range queries) and those that are accessed by different types (e.g. once accessed through point and range queries). The motivation behind this approach is, that we can use this information to optimize our hybrid index structure such that the underlying data structure is optimized for certain sub-ranges. One optimization that comes to mind is the use of a hash table for a segment that is only accessed through point queries. One can benefit from the faster lookup time while the disadvantage of hash tables, the unsortedness of the keys, has no impact because we know that we have no range queries that access this segment.

Similar to the frequency algorithm above, we ruled out the use of key-by-key comparisons because of the generalization problem. Instead, we used a rather simple algorithm that tries to find the boundaries where the query type changes. The goal here is the determine partitions that have pure access patterns, so one partition should mostly contain one query type (or only mixed accesses).

The algorithm is a single-pass over the data, where we again consider a sliding window around the currently selected key. For each key, we store the predominant query type in that window, and as soon as we have a different major query type than for the previous key, we begin a new partition.

4.4 Index Bulkloading and Benchmarking

To understand how we incorporate the information of the partitioning into our index, we need to cover the general structure of the hybrid index and how it is built and bulkloaded before being benchmarked. Additionally, we look at how the index is altered by the partition information.

4.4.1 Structure of the index

The general structure of our index is very similar to the indexing framework presented by Dittrich, Nix and Schön [12]. Our index has the same internal structure as a B^+-tree, but we do not fix the size of the leaf nodes. Instead, each leaf is designed to represent exactly a partition produced by the partitioning algorithms from before. As described in Section 3.2, we have the flexibility to choose the data layout and search strategy inside each node separately. The default search strategy for the internal nodes (to locate the next child) and also the leaf nodes (to locate the final position) was chosen to be binary search, as we deal only with sorted data in this work. As mentioned

Algorithm 2 Partition by Purity

```
idx ← 0
n ← data.size
while idx < n − w do
    current_freq ← freq[idx]
    fwd_mean ← mean(freq[idx : idx + w])
    fwd_slope ← fwd_mean/w
    if isclose(fwd_slope, 0, delta/w) then
        potential_start ← idx
        idx ← idx + 1
        for i in 1..w//2 do
            central_left ← mean(freq[idx − i : idx + 1])
            central_right ← mean(freq[idx + 1 : idx + w//2 + 1])
            central_slope ← (central_right − central_left)/(w/(2 + i + 1))
            if !isclose(central_slope, 0, delta/w) then
                idx ← potential_start
                break
            end if
        end for
        if idx == potential_start then
            idx ← idx + 1
            continue
        end if
    end if
end while
```

before, unsorted data poses some challenges regarding the routing information inside our hybrid index, which makes it difficult to correctly identify where a key is located in the leaves.

4.4.2 Changing leaf data structure

The first way of optimizing our index given the partition information, is to change the leaf data structure that is used to map the keys of our dataset to their position. The default choice of a binary search with a sorted layout is well suited for range queries, as we only need a lower bound point query and an additional scan until the upper bound afterwards to determine all keys that qualify for for the query. This is a sensible default, but for a point query only segment, we choose to use a hash table instead. As mentioned before, we achieve a better lookup performance for point queries, and as there are (almost) no range queries in the segment, we do not need to support an efficient lookup for them. Should we encounter a range query in the test workload, we can still guarantee the correctness of the results by converting the range query to a series of point queries that can be executed on the hash table. While this optimization seems straight forward, we also have the possibility to change the data structure of the leaves for other different scenarios, although that was not done in this work.

4.4.3 Moving leaves higher up

The next way of optimizing the index is to utilize the frequency of the partitions that are generated more effectively. Apart from only creating the partitions, one could also consider to improve access times for frequently visited segments. One way of doing so was presented in Section 2, where the authors adaptively classified nodes as hot or cold based on current and past access statistics. Then they either applied a peformance-optimized or space-optimized encoding to these nodes, depending on the classification. Another approach that we wanted to look into is moving the leaf segments with a relatively high freuqency higher up in the tree, which would result in a shallower path to the corresponding segment in our index. We suspect that we could benefit from caching effects here, as there are less nodes above the leaf segment that need to remain in cache for faster access.

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

Clear cuts experiment First of all, we conducted a rather simple experiment that was designed to confirm the result of Dittrich, Nix and Schön [12]. While they hand crafted the boundaries, the partitions for the experiment here were generated by the purity partitioning algorithm. The workload that

was executed on the data was split into three regions: the first 10 percent of the keys, the next 75 percent and finally the remaining 15 percent. The first region received 20 percent of the total queries as point queries, the second region received 10 percent of the queries as point queries and 20 percent as range queries. The final region received the remaining 50 percent of queries as point queries. For our experiment, we used 2 million queries in total of which 1 million served as a train workload and 1 million as a test workload. For the benchmarking on the test workload we therefore had the queries depicted in Table 5.1.

Region	from	to	number queries	query type
1	0	0.1	200k	Point
2	0.1	0.85	100k	Point
2	0.1	0.85	200k	Range
3	0.85	1	500k	Point

Table 5.1: Query Distribution for clear cuts experiment

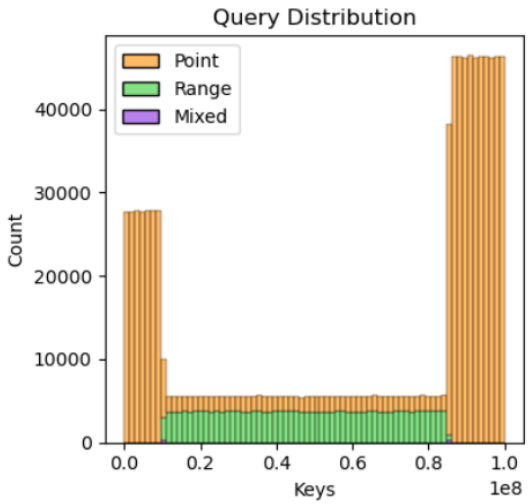

Figure 5.1: Query Distribution of clear cuts experiment over 100 million uniform dense keys

The resulting query distribution is visualized in Figure 5.1. As we could already tell from the description of the query distribution, we have two very clear boundaries when looking at frequency as well as purity. While the first and last region are only requested by point queries, we can see that the middle part receives both types of queries. Notably, we see mostly orange and green bars in the figure in this region, as this range contains 75 million keys, but receives only 300 thousand queries. While there is some overlap

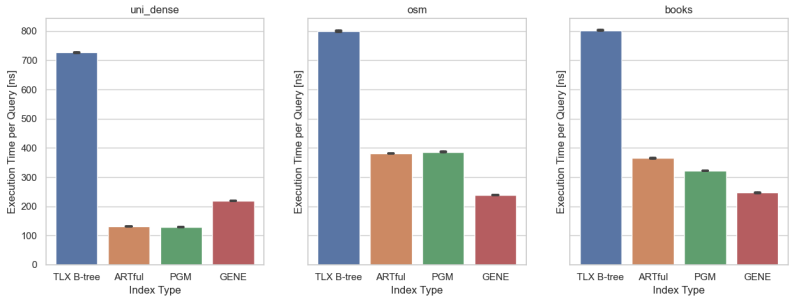

Figure 5.2: Average Query Execution Time on uniform dense data and the osm and books dataset

of these queries, it is rather unlikely that one key from this range is selected for both point and range queries. This overlap would result in a purple bar in the figure. Nevertheless, a generalizing partitioning algorithm should recognize the boundaries and create three partitions accordingly.

//TODO: too small, make this bigger, maybe split single plots

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] 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.

[13] 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.

Appendix A

Appendix