Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bachelorarbeit

Learned Index Structures: An Evalution of their Performance in Key-Value Storage Solutions

vorgelegt von

Leonhard Reichenbach

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Computing in Science Sp. Physik
Matrikelnummer:	6689683
Erstgutachter:	Dr. Michael Kuhn
Zweitgutachter:	Jakob Lüttgau
Betreuer:	Dr. Michael Kuhn, Jakob Lüttgau

Hamburg, 2018-09-10

Abstract

Traditional index structures like B-Trees and hash tables have reached the peaks of their performance. Their existing implementations are highly tuned to use the hardware as efficiently as possible. However, they are static solutions and do not exploit existing inner structures in the indexed data which would offer another source for further performance increases. Additionally, some of their underlying techniques are fundamentally flawed. For instance, the hash function in a hash table that hashes keys to uniformly distributed random numbers will always inevitably cause a collision rate of approximately 36% due to statistic phenomena. Thereby, creating a demand for better solutions.

One of those solutions could be learned index structures. The traditional index structures are in the wide sense just models taking a key and predicting an index position. If the resulting array is sorted, the index is effectively modeling a cumulative distribution function. If this distribution can be learned by a machine learning model like a neural network, the potential to reduce collisions further is given.

In this thesis, an in-memory key-value store named HTableDB was implemented. It is able to use learned index models as well as regular hash functions for indexing and thus offers an easy way to compare the two. Using HTableDB this comparison was made for keys consisting of various file paths taken from a scientific computing environment. The learned index model used was a two-stage feed-forward network. For datasets with keys that belong to a cluster of similar ones, like the ImageNet dataset, the learned index models showed a collision rate of 43% while being approximately 45 times slower than regular hash functions. However, more sophisticated network architectures could lower the collision rate of learned index models in the future.

Acknowledgments

I thank Dr. Michael Kuhn and Jakob Lüttgau for their helpful advice and the proofreading. Furthermore, I thank Kira for never getting tired of correcting the same spelling mistakes again and again.

Contents

1. Introduction

This chapter explains the goals of this thesis and the motivation behind them. First, the importance of modern high-performance computing (HPC) and the need for better index structures is elaborated. Afterwards, the goals of this thesis are defined. Finally, the thesis outline is presented.

1.1. Motivation

The rapid technological progress of the last century has impacted society in a way comparable to the discovery of fire, the first domestication of agricultural plants and the invention of the steam machine. Nowadays, the vast majority of the human population carries a smartphone, enabling them to communicate with others, take pictures and videos, show their exact location and velocity, the current time and an accurate weather forecast. On large supercomputers physical experiments can be simulated and medical diagnoses have improved to levels previously unthought of. These developments are fueled by the ever-improving performance of computer systems, founded on the invention of integrated circuits.

Following Moore's Law, the number of transistors in dense integrated circuits doubles every 18 months, increasing computational power at a rapid pace. This rapid increase leads to the emergence of ever-new technologies and computational approaches as the implementation of once purely theoretical concepts becomes feasible. An area benefiting from this is the field of artificial intelligence (AI), neural networks and deep learning in particular. As deep learning becomes cheaper due to the increase in computational power, new applications using it keep appearing, solving previously hard problems with ease. These problems include, but are not limited to, human-like tasks like image recognition, speech synthesis, automatic translation and medical diagnosis.

With increasing computational power the need to store and access data increases as well. To access data, index structures are essential. The commonly found index structures like B-Trees or hash tables are general purpose data structures not making any use out of a possible internal structure of the stored data. For example, when storing n fixed-length entries having continuous integers as keys in a conventional B-Tree the lookup time scales with $\mathcal{O}(\log(n))$, whereas the keys itself could be used as an offset, shrinking the lookup time to $\mathcal{O}(1)$ [Kra+18]. Hash tables already achieve $\mathcal{O}(1)$ lookup times. However, the used hash functions do inevitably suffer from collisions as they distribute values in an independent uniform manner. Those collisions cause the hash table's performance to degrade and, depending on the hash table implementation, waste memory.

One of the key strengths of neural networks is their ability to learn structures hidden in data. If a neural network can perfectly learn the data's distribution and successfully generalize it to similar data, it would have zero collisions while maintaining an access time of $\mathcal{O}(1)$. Therefore, the approach to enhance traditional index structures using learned models is taken.

1.2. Related Work

Kraska et al. analyzed the performance of learned indexes in comparison to B-Trees, hash tables and Bloom-Filters with promising results [Kra+18]. They used a concept they called the *recursive-model index*, consisting of multiple stages of layered networks. For hash tables, they found a collision reduction of up to 77% when using integer keys. This was achieved by learning the *cumulative distribution function* (CDF) of the keys. Additionally, a string dataset was tested and the index model was compared against B-Trees resulting in less memory used and, in some cases, faster lookup time.

1.3. Thesis Goals

The goal of this thesis is to evaluate and compare the performance of learned index structures to traditional methods. In particular, the hash table data structure will be analyzed and its hash function will be replaced by a learned index model. The goal of this replacement is to reduce the rate of collisions and thus the amount of space wasted by them. Additionally, the index model's execution speed will be compared to regular hash functions to decide if the replacement is feasible in practice. The combination of index models and hash tables integrated into a key-value store and evaluated for its applicability for storing file system metadata. Thereby, the keys to calculate the indexes from will be strings, in particular, file paths.

1.4. Thesis Outline

Chapter 1 motivates this thesis and defines its goals. Chapter 2 introduces the necessary background knowledge of local and parallel file systems, key-value stores, hash tables and hashing in general as well as neural networks. Chapter 3 explains the approach taken as well as the design choices for the key-value store HTableDB.

Chapter 4 contains implementation details of HTableDB and its subcomponents. In Chapter 5, several aspects of those components are evaluated. Chapter 6 concludes this thesis and presents an outlook on future work regarding learned index structures.

Summary

This chapter explained the shortcomings of traditional index structures and introduced learned index models as a potential way to solve those issues. Additionally, the thesis goals of comparing learned index models to regular hash functions were defined. Finally, the structuring of the thesis was outlined.

2. Background

This chapter provides the necessary background knowledge of storage systems, data structures and machine learning. First, the concept of local file systems and metadata is introduced. Then the scope is widened to parallel file systems and JULEA, a flexible storage framework. Afterwards, key-value stores, hash tables and (fixed-length) hashing approaches are elaborated. The chapter concludes with an overview of neural networks and their training, followed by a brief explanation of linear regression.

2.1. Local File Systems

In Section 1.3 file system metadata and file paths were mentioned, to understand both, first, the concept of file systems is needed. Local hierarchical file systems, for example, ext4 or XFS, are used to manage and store data on local block devices like hard disk drives (HDDs) or solid state drives (SSDs). Instead of having to deal with raw memory addresses and offsets, they present a tree-like structured overlay to the user. The structuring is done by using directories and files constituting the essential components. Files are named blocks of data ranging from simple text files to complex executables. Directories, on the other hand, are a special kind of file containing other files and (sub)directories. Directories do not physically contain the data of the files residing inside of them, instead, they contain information referencing those files. Thereby, a tree-like structure, the so-called directory tree, with the root directory `/` at the top, is formed. In Figure 2.1 a small example directory tree is shown.

Every file in the file system is part of the directory tree and can be identified by its path. The path is a unique identifier, meaning that an absolute path is only pointing to one specific file on the file system. A file can be referenced by more than one path to be made available in different parts of the file system. The path is the textual representation of a walk through the directory tree to reach the file. In Figure 2.1 the path to the file `file` is shown. The path consists of three parts, the names of the root directory `/`, the subdirectory `dir/` and the file `file`. This path layout is common through most file systems as it is part of the *portable operating system interface* (POSIX) standard which Unix-like operating systems adhere to. The length and the character set of a path can be limited on certain systems but on

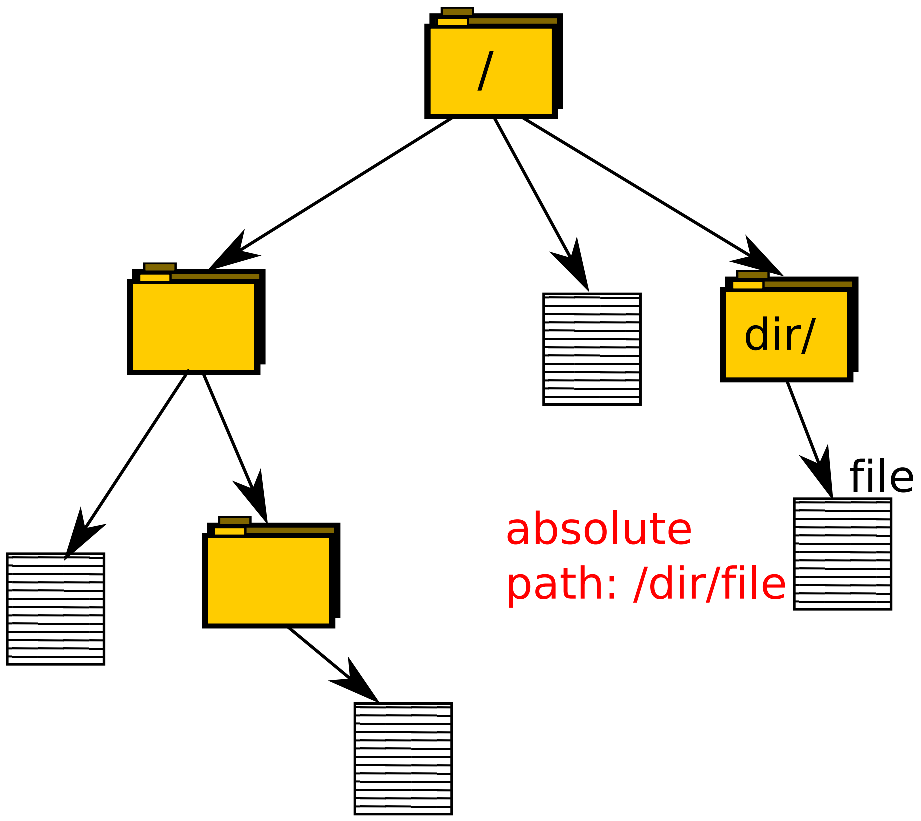


Figure 2.1.: A directory tree

modern systems, a path can usually be of arbitrary length containing any Unicode character.

To achieve its task of managing and storing data the file system requires an amount of additional information, like the file's access permissions, the ID of the user that owns it and, most importantly, the file's size and location on the block device. This is the so-called metadata. The metadata of a file is saved in a so-called inode, an object not directly visible to the user. Table 2.1 shows the structure of an ext4 inode. It stores the aforementioned metadata and the data's location in a 256-byte small structure. The inodes are stored at the beginning of the block device and need to be accessed first before the actual data of file can be accessed. If the file is smaller than 60-bytes it is stored in the inode itself.

2.2. Parallel File Systems

Modern high-performance computing requires massively parallel compute architectures with thousands of compute nodes as described in Section 1.1. The data generated by those computations is in the range of petabytes which is too much to fit on

size	**content**
2 bytes	permissions and type
2 bytes	lower 16-bits of owner UID
4 bytes	lower 32-bits of size in bytes
4 bytes	last access time
4 bytes	last inode change time
4 bytes	last data modification time
4 bytes	deletion time
4 bytes	lower 16-bits of group ID
⋮	⋮
60 bytes	block map, extent tree or inline data
⋮	⋮
4 bytes	project ID
96 bytes	free space for extended attributes
256 bytes	**overall**

Table 2.1.: ext4 inode [Djw17]

a single storage server with a local file system. To keep up with the resulting I/O needs of such systems the parallel file systems were created. Parallel file systems can store data distributed over a vast amount of block devices located in several servers, aggregating their capacity. Compute nodes request access to this data through local clients over a network connection to the storage servers of the parallel file system. Large files can be split up into multiple stripes residing on different servers to aggregate the performance of multiple storage devices to achieve maximum throughput for the clients. This is necessary, as the interconnect throughput excels the drive throughput by a factor of up to 3000[1].

Additional metadata is required to keep track of where a file is stored, as the mapping of file stripes to servers becomes essential. At scale, the differences between accessing data and metadata come to play. Data is usually "big" and can be read in a continuous stream, while metadata is small and needs to be randomly accessible. As the parallel file system needs to first access the metadata of a file because it contains the data's location, metadata and data are commonly stored separately to enable faster accesses. To achieve this, parallel file systems use separate servers dedicated only to metadata. The metadata servers (MDS) can be outfitted with faster storage devices like (NVME) SSDs, more memory for caching and powerful CPUs to handle large numbers of concurrent accesses. Data servers, however, can be fitted with HDDs as they mainly deal with large chunks of data. HDDs are better suited for

[1]For instance, the 100 GBit/s data rate of Infiniband EDR in contrast to the 255MB/s sustained transfer rate of a HGST Ultrastar DC HC520 12TB HDD

this case as they deliver a higher capacity for less money than SSDs. This enables sufficient file system performance while keeping the costs acceptable [Lüt+18].

A typical access of a client to a file consists of multiple steps. First, the file systems management server is contacted to gather the information which server to contact next. Then, the right metadata server is contacted to find out on which data servers the file is located. Afterwards, the corresponding data servers are accessed in parallel to maximize throughput.

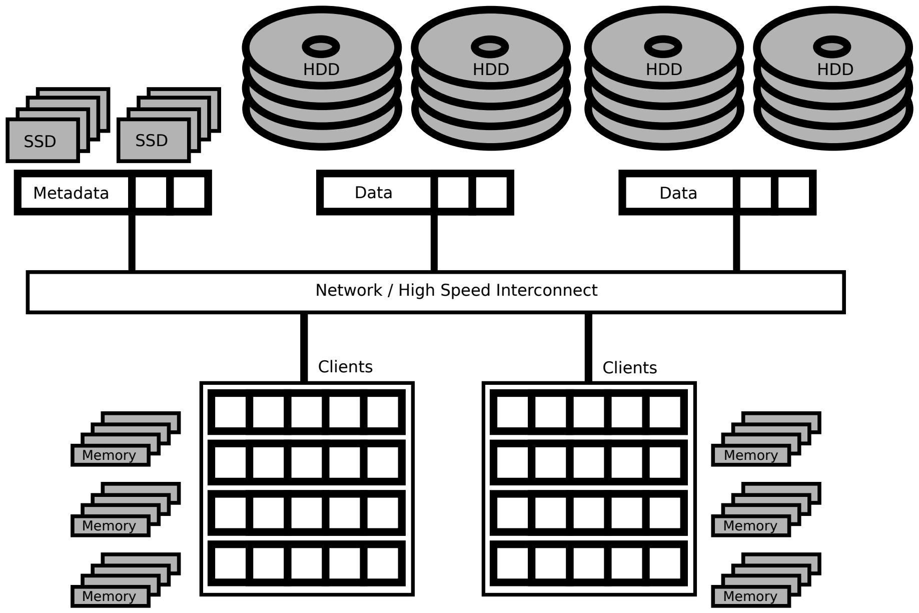
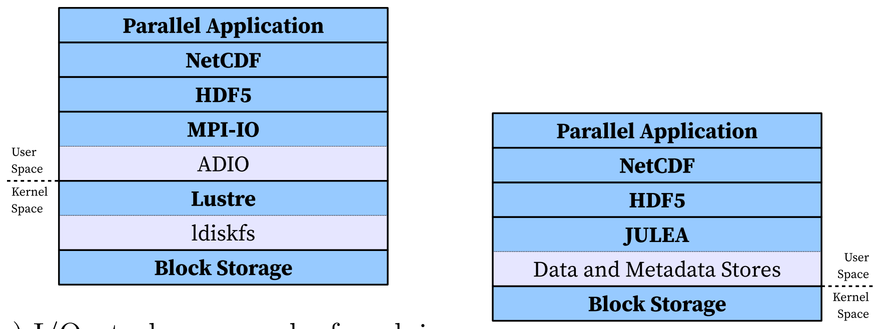


Figure 2.2.: Schematics of a parallel file system

2.3. JULEA

Parallel file systems as introduced in Section 2.2 are complex and hard to extend further. However, most scientific software already uses data description frameworks such as the NetCDF [Uni18] and HDF5 [The18] libraries for I/O. Thus, they do not need all of the features offered by a classic parallel file system and could profit from simpler storage systems offering a direct interface to those libraries [Lof+16].

JULEA [Kuh17] is such a flexible storage framework offering dynamic I/O semantics as well as interchangeable interfaces and backends as needed in the future. The proposed I/O stack difference is shown in Figure 2.3. It is intended to be a platform for teaching and academic research of future I/O solutions. JULEA is written completely in userspace C which, combined with the modular backend design, enables students and researchers to rapidly prototype and test their ideas. In contrast, parallel file systems like Lustre require changes to kernelspace code to implement new features



(a) I/O stack commonly found in HPC

(b) Proposed I/O stack with JULEA

Figure 2.3.: I/O stack differences
Taken from [Kuh17]

which makes debugging and testing more complex. Following the established concept JULEA also differentiates between data and metadata. The data is saved in an object store, while the metadata is put in a key-value store like Leveldb [GD] or LMDB [Sym]. For this thesis, the focus lies on the key-value stores.

2.4. Key-Value Stores

A key-value store offers a simple yet efficient way of managing (storing and accessing) arbitrary data. The fundamental working principle behind a key-value store is the associative array, also known as the dictionary. Associative arrays reference their entries by a key e.g. a naming string or an integer id. In key-value stores, the entries also referred to as values are opaque blocks of data from the system's point of view with the possibility to have a different internal structure for every entry. This is the major difference in contrast to relational databases where data is stored in the predefined fields of various tables. While relational databases have the possibility to use the well-defined structure of the stored data to apply optimizations, key-value stores offer a simpler interface and additional flexibility. Basic key-value stores can store their data in an unordered fashion, more complex ones offer features such as storing the data sorted by the keys. This mostly depends on the choice of index structure for the array, e.g. hash tables in contrast to trees. Key-value stores are a widely used concept and most programming languages either provide associative arrays as a primitive type (e.g. Python dictionary) or via libraries (e.g. C++ std::map, Java Map). However, until recently, key-value stores were rarely deployed at large-scale as relational databases were and are still better at handling traditional workloads dealing with relationally structured data like e.g. managing customer records. Today, key-value stores are used almost anywhere, from large cloud providers and hyperscalers (Amazon DynamoDB [AWS]) to the local IndexedDB in web browsers (LevelDB in Google Chrome [Chr]). Implementations exist in a wide variety of different persistence and consistency levels, from in-memory variants like Memcached [Fit+] to persistent ordered ones like LMDB [Sym]. For this thesis, the most important aspect of key-value stores is the already mentioned simplicity of the interface. Using a key-value store only requires the following three basic functions:

- `put(key, value)`
- `get(key)`
- `delete(key)`

To store a key-value pair `put(key, value)` is used. `get(key)` returns the value for a given key, provided that such a key-value pair was previously stored. Finally, `delete(key)` is used to delete the corresponding key-value pair from the store.

2.5. Hash Tables

Hash tables are key-value stores. They store key-value pairs in an array also referred to as *table*. The table's slots are called buckets. The position p of a key-value pair's bucket is computed by an index function

$$p = f(k, n) \tag{2.1}$$

where k is the key and n is the array's size. This is usually done by hashing the key k with a hash function which turns it into a scalar value. Afterwards, this value is shrunk to get a valid array index. This is achieved by using the modulo operation. Hence, f is of the form

$$f(k, n) = h(k) \mod n \tag{2.2}$$

where h is a hashing function [Knu98]. Hash functions will be explained with further detail in Section 2.6.

The amount of time needed to compute a key's hash does not increase with the number of entries stored by the hash table. Thus, an entries lookup time is constant and the time-complexity of the three basic operations, in the average case, is $\mathcal{O}(1)$. This is the major benefit, in contrast to, e.g. the time-complexity of $\mathcal{O}(\log(n))$ when the indexing is done with B-Trees.

Table 2.2 shows a comparison of the time-complexity of B-Trees and hash tables in the average as well as in the worst case. To explain the hash table's complexity of $\mathcal{O}(n)$ in the worst case the concept of collisions is needed. Hash functions can output the same hash for two (or more) different keys, thus, the hash table needs a mechanism to handle these so-called (hash-)collisions. More details regarding the

	Hash table	B-Tree
Average	$\mathcal{O}(1)$	$\mathcal{O}(\log(n))$
Worst	$\mathcal{O}(n)$	$\mathcal{O}(\log(n))$

Table 2.2.: Time-complexity of hash tables and B-Trees

mathematical background of collisions will be given in Section 2.6.1. In the worst case, the index function maps all keys to the same bucket, thus, a linear search is required to find the right key-value pair. An important measure when talking about hash tables and collisions is the load factor α defined by

$$\alpha = \frac{m}{n} \tag{2.3}$$

where m is the number of entries in the hash table and n is the total number of buckets. When assuming that any given key has equal probability to get mapped to each of the n buckets, independent of where other keys are mapped to, the load factor α is also the average number of elements mapped to the same bucket.

The previous assumption is called *simple uniform hashing* [Cor+09]. Under this assumption, the hash table's operations should maintain their excellent time-complexity up to a load factor of one. In practice, however, the hash table's performance starts to degrade before reaching that. The reason for this is the unexpectedly high collision rate explained in Section 2.6.1. A common approach to maintain a hash table's performance is to increase its size before the load factor gets close to one. For instance, in Google's `dense_hash_map` [Goo05], by default, the table size is doubled after reaching a load factor of 0.5. This thesis only considers fixed size hash tables to compare hash functions with index models without impacts caused by the resizing.

2.5.1. Separate Chaining

To deal with collisions different approaches exist. In the following, two of them will be discussed further, namely separate chaining and open addressing. In separate chaining, every bucket of the hash table contains a pointer to a corresponding list structure. The elements that are mapped to this bucket are appended to its list. These lists are called the hash table's chains commonly implemented using simple linked lists.

When a collision occurs the additional, colliding, key-value pair gets added to the end of the bucket's list. Here, the load factor α also corresponds to the average number of elements per chain, also referred to as the average chain length. When assuming simple uniform hashing, the search for a key has a time-complexity of $\Theta(1 + \alpha)$, in the average case. With this approach, hash tables can be overloaded to contain more elements than buckets. Even when a hash table is massively overloaded its

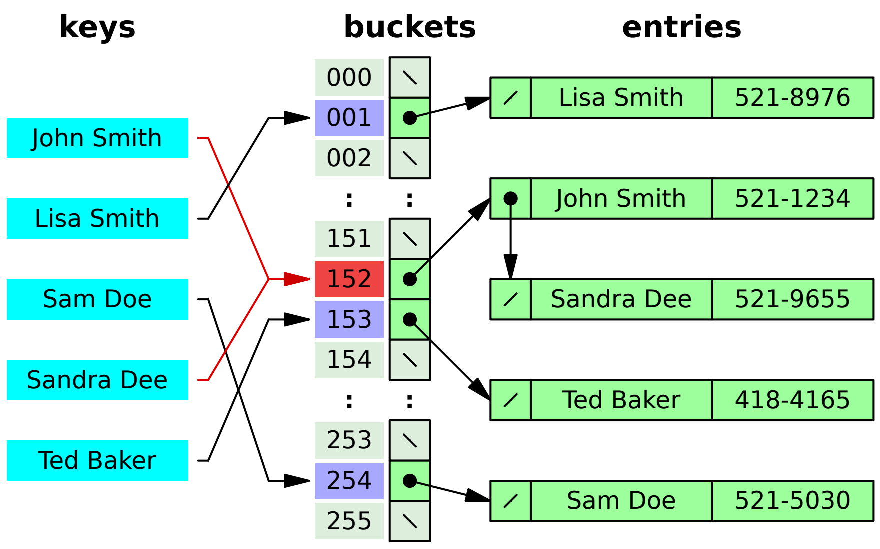


Figure 2.4.: A hash table using separate chaining with simply linked lists. The red arrows indicate colliding entries.
Taken from [Sto09c]

performance is still sufficient as on average the list's length is still small compared to the table's size. So, only a small part of the elements must be searched to find the right one. The ability of overloading can make later resizing of the hash table unnecessary. However, the traversal of linked lists can be slow as the entries usually are non-contiguous in the memory leading to potential cache misses.

To explain what cache misses are and how they are impacting an operation's execution time, first, an overview over a computer's memory architecture is necessary. The actual operations in the processor are done on values residing in the processor's registers, a really fast type of memory located directly next to the processor's other components on the chip. However, the data that the processor is supposed to operate on is located in the main memory located on different dedicated memory chips as the registers are too small to fit the necessary amounts of data. Furthermore, the main memory is made out of a different type of memory called DRAM, which is cheaper and can be packed tighter. This difference enables the large amounts of main memory needed for current programs. For instance, the number of registers for floating point operations in a core of a processor supporting *Advanced Vector Extensions* is 16, each having a capacity of 256 bits for a total of 512 bytes respectively 1024 bytes in a dual-core processor like the Intel® Core™ i5-4210U. However, the same processor supports up to 16 GB of main memory. The DRAM used for the main memory is slower than the registers. This is mitigated by several layers of caches. Caches are made from another type of memory SRAM and are also located on the processor's chip.

When a part of the computer's memory is accessed by the processor, a fixed-size region of the memory is stored in the cache. This cache entry is called a cache line. Subsequent processor accesses to the same portion of memory are now faster. If none of the stored cache lines contain the wanted portion of memory a cache miss occurs. A cache line is evicted from the cache when the place is needed for another portion of memory. To decide which cache entries should be evicted the processor can use different strategies. Two of those strategies are to either remove the *least recently used* (LRU) entry or the *least frequently used* (LFU) entry. As the bucket array is contiguous in memory and frequently accessed, it is likely stored in the cache if it fits [AA15].

When one of the hash table's entries is searched, first, its position in the bucket array is accessed. This position contains a pointer to the head of that bucket's list and is cached if the array is cached. However, it is likely that the list entry itself is not cached. If this is the case a cache miss occurs and the data needs to be transferred from the memory, which takes an additional time of up to 100 ns, depending on the hardware. For instance, for an Intel® Xeon® Processor E5-2699 v4, the difference in access time is 75 ns [7CPU]. The probability of the returned list entry to be the searched one is only $^1/_\alpha$, likely making it necessary to traverse to the next list element which again can lead to a cache miss. Additionally, the chance of a list element to be in the cache decreases when traversing the chain as elements located further to the end are less likely to be accessed during previous searches for other elements. This leads to high search times. Thus, a hash table using separate chaining requires a size such that the load factor is low.

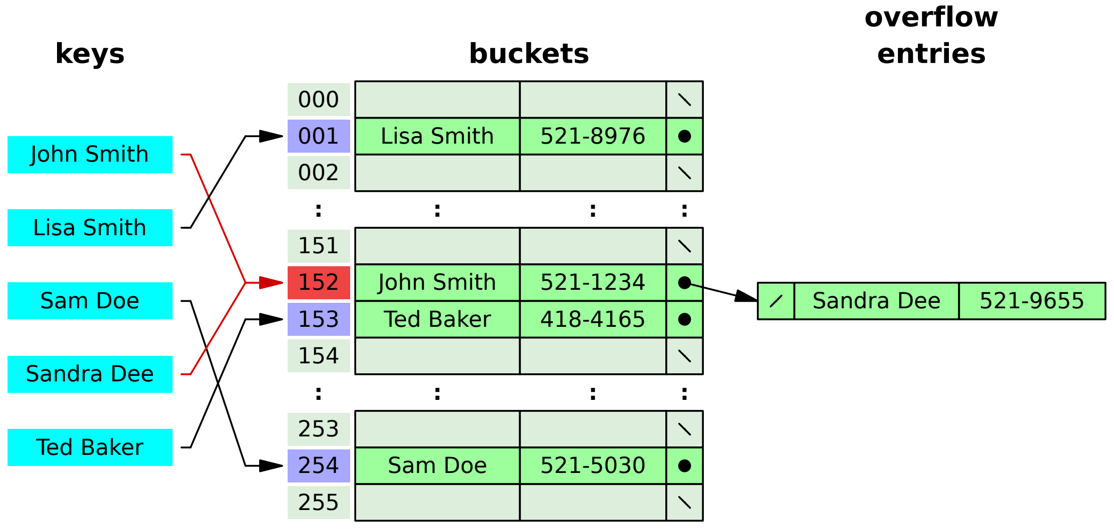


Figure 2.5.: A hash table using separate chaining with head entries. The red arrows indicate colliding entries.
Taken from [Sto09a]

To decrease the number of pointer traversals, and thereby the number of cache misses, the list's head records can be directly stored in the bucket array itself. Ideally, no list traversal is needed in this case. Figure 2.5 shows such a hash table and the according collision handling. The disadvantage of this approach is that empty buckets use more space than just a pointer to a list. Over-allocating the bucket array to guarantee a low load factor can become unfeasible as the memory need for the initial bucket array can can increase by a factor of two or more. The memory requirement doubles if a pointer to a container structure containing key and value is stored in addition to the list pointer; assuming all pointers are of the same size. It triples if instead of a container pointer, the pointers to key and value are stored directly in the buckets. If keys and/or values are fixed in size they can be placed directly into the buckets, decreasing the number of pointer traversals even further while increasing the penalty for empty buckets.

2.5.2. Open Addressing

Similar to the previously discussed approach of separate chaining with the head entries in the table's buckets, the concept of open addressing stores the key-value data in the array itself. If the bucket returned by the index function is already taken another bucket is picked during insertion. The choice complies with a probing scheme like linear or quadratic probing or double hashing. In linear probing, the next bucket chosen has a fixed (linear) offset to the initial bucket. In contrast, in quadratic probing, this offset is increased quadratically. Double hashing scales the offset with a value computed by a second hash function.

In the following, linear probing with a probing interval of one is used. The next free bucket when storing a colliding entry is therefore acquired by linear search. When the position of a previously stored element is needed to get or delete a key-value pair and the position returned by the index function for said element does not match it is necessary to search the next buckets in the array. The search continues until the entry with the desired key is found.

This scheme introduces additional collisions for keys with distinct hashes when reaching a certain load factor as keys get mapped to buckets already used by the overflow of the previous collisions. An example of this is shown in Figure 2.6, where the entry with the key ”Ted Baker” collides with the entry with key ”Sandra Dee” despite being mapped to different array positions. The additional collisions and increasing distances between mapped and real bucket further increase the time needed for the hash table operations. Fixed size hash tables using open addressing cannot be overloaded as it is not possible to store additional entries once the bucket array is full.

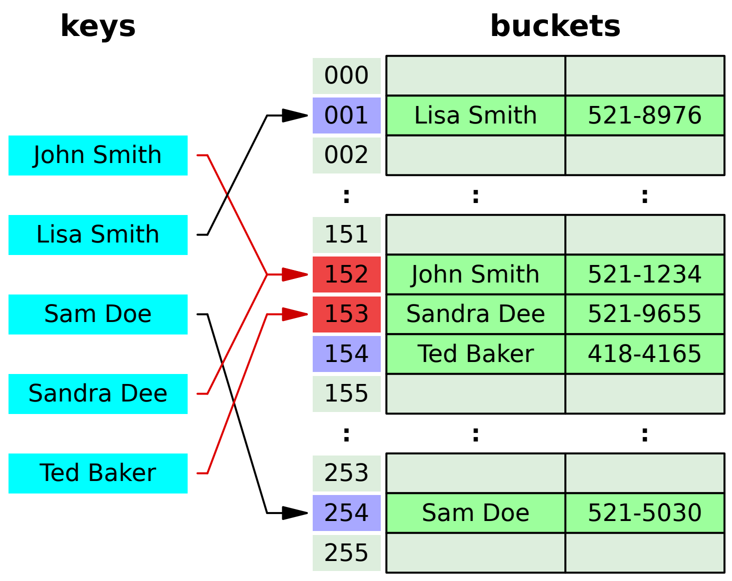


Figure 2.6.: A hash table using open addressing with linear probing. The red arrows indicate colliding entries.
Taken from [Sto09b]

2.6. Hashing and Index Functions

A good hash function should satisfy two requirements:
a) Its computation should be really fast.
b) It should minimize collisions.

Knuth

In Section 2.5 a distinction was made between hash functions and index functions. Books like *The art of computer programming, Volume III, 2nd Edition* and *Introduction to Algorithms, 3rd Edition* call the whole function mapping a key to an index the hash function. However, in practice, hash table implementations commonly use the modulo reduction approach shown in Equation (2.2), where the key is hashed by a function and then reduced to fit the table. For this reason, in this thesis, the distinction is made and the outer function is called index function.

An index function is a function

$$f\colon K \times \mathbb{N} \to I \tag{2.4}$$

where K is the keyspace and $I = \{i \in \mathbb{N} | i < n\}$ is the set of possible indices for an array of size n, which maps keys $k \in K$ to array indices $i \in I$. This function neither needs to be injective or surjective, in particular, this means that the trivial index function

$$f_t \colon (k, n) \mapsto 0, \tag{2.5}$$

mapping every key to the first slot of the array, is an index function. When the keyspace K and the index space I have the same number of elements ($|K| = |I|$), a bijection exists mapping keys to indices with no collisions and no empty array slots left, this is called a (minimal) perfect index function. In practice, such a function can only be found when all possible keys to be stored are already known beforehand. This is called static hashing [FKS84]. Another approach, called dynamic perfect hashing, exists working with subtables where parts of the hash function are altered if a collision occurs. Afterwards, the corresponding subtable is rebuilt [Die+88].

A hash function h is the part of an index function f that reduces the possibly multi-dimensional key k to a scalar value so that

$$f(k) = s(h(k), n) \tag{2.6}$$

where s is a function that scales the output of h so that it fits into I. In this thesis two variants of s are used

$$s_1(h(k), n) = \lfloor n * h(k) \rfloor \tag{2.7}$$

$$s_2(h(k), n) = h(k) \mod n \tag{2.8}$$

where $n = |I|$. When h maps to $\{x \in \mathbb{R} | 0 \leq x \leq 1\}$ s_1 is used to scale up the value to fit the elements of I. When h maps to $\{x \in \mathbb{N} | x \leq \texttt{INT\_MAX\_VALUE}\}$ s_2 is used to reduce the value to fit the elements of I. The first case is needed when using neural networks instead of a classic hash function. The reasons for this differentiation will be explained in Section 2.7. The latter case is used with classic unlearned hash functions.

2.6.1. Collisions

In Section 2.5, the performance impact of collisions on hash tables was explained. In this section, the collision rate of the index function will be analyzed further.

Assuming simple uniform hashing, every key k_i has the same probability to get mapped to any of the n buckets. Thereby, the probability $P(A)$ of the event $A \colon f(k_i, n) = j$ that a key k_i gets mapped to a bucket b_j is

$$P(A) = \frac{1}{n} \quad \text{and the complement is} \tag{2.9}$$

$$P(A^c) = 1 - P(A) = 1 - \frac{1}{n}. \tag{2.10}$$

The probability of the event $B\colon f(k_i, n) = f(k_\ell, n) = j$ that another key k_ℓ is mapped to the same position is given by

$$P(B) = 1 - P(A^c)^m. \tag{2.11}$$

where m is the number of keys added to the hash table. When $m = n$, as in the case of $\alpha = 1$, Equation (2.11) is

$$\begin{aligned} P(B) = 1 - P(A^c)^n &= 1 - (1 - P(A))^n && (2.12)\\ &= 1 - (1 - \frac{1}{n})^n \quad \text{which converges fast to} && (2.13)\\ \lim_{n\to\infty} 1 - (1 - \frac{1}{n})^n &= \frac{-1 + e}{e} \approx 63.21\% && (2.14) \end{aligned}$$

However, this is not the probability of a collision to occur in the whole array. This is only the probability of two of the n keys colliding in a specific bucket. It is a common mistake to confuse these two events and, thus, their probabilities. This is called the birthday paradox.

To calculate the probability for the event $C\colon f(k_i, n) = f(k_\ell, n) \forall i, \ell \in \mathbb{N}_{<m}\colon i \neq \ell)$ that a single collision occurs when inserting m keys into the hash table, a different approach is needed. Let u be the number of possible non-colliding key combinations and g the number of all possible key combinations. Then one has

$$\begin{aligned} u &= \frac{n!}{(n-m)!} \qquad\qquad g = n^m && (2.15)\\ P(C^c) &= \frac{u}{g} && (2.16)\\ \Rightarrow P(C) &= 1 - \frac{u}{g} = 1 - \frac{n!}{(n-m)! \cdot n^m} && (2.17) \end{aligned}$$

$P(C)$ is the probability that at least one collision occurs. Let now $n = 1000$, then for $m = 38$, $P(C) \approx 50\%$. Although only 38 entries were added to the hash table the probability that a collision occurs is already higher than 50%.

The total number of collisions when inserting a fixed number of entries into the hash table can also be calculated. Let $i - 1$ be the number of entries already in the hash table. Then the probability that a key k_i does not collide with any previous key is

$$\begin{aligned} P &= \left(\frac{n-1}{n}\right)^{i-1} \quad \text{and} && (2.18)\\ P^c &= 1 - P = 1 - \left(\frac{n-1}{n}\right)^{i-1}. && (2.19) \end{aligned}$$

As each new entry can either collide or not P^c is also the average number of collisions added by a key k_i. To get the total number of collisions N the sum over all $P^c(i)$ is taken

$$N = \sum_{i=1}^{m} \left(1 - \left(\frac{n-1}{n} \right)^{i-1} \right) \tag{2.20}$$

$$= m - \sum_{i=1}^{m} \left(\left(\frac{n-1}{n} \right)^{i-1} \right) \tag{2.21}$$

$$\text{geometric series} \Rightarrow N = m - n + n \left(\frac{n-1}{n} \right)^{m} \tag{2.22}$$

Furthermore, if $m = n$, Equation (2.22) reduces to

$$N = n \left(\frac{n-1}{n} \right)^{n} \tag{2.23}$$

$$\Leftrightarrow \frac{N}{n} = \left(\frac{n-1}{n} \right)^{n} \quad \text{which quickly converges} \tag{2.24}$$

$$\lim_{n \to \infty} \frac{N}{n} = \lim_{n \to \infty} \left(\frac{n-1}{n} \right)^{n} \tag{2.25}$$

$$= \lim_{n \to \infty} \left(1 + \frac{-1}{n} \right)^{n} \tag{2.26}$$

$$\overset{\text{def}}{=} \exp(-1) = \frac{1}{e} \approx 0.367 \tag{2.27}$$

In conclusion, when a hash table has a load factor of one, 36% of the entries caused a collision. Moreover, if a hash table with separate chaining is used as described in Section 2.5.1, 36% of the buckets are remaining empty as the colliding entries are stored in the chains. This is a waste of space. Therefore, a need arises for index functions that distribute the data in a more efficient way than with simple uniform hashing.

2.6.2. Computation of Hash Functions

All possible keys k can be expressed as byte arrays $k = \{b_0, \ldots, b_{m-1}\}$. To map the whole array to a single value the hash function needs to condense the array. This can be as simple as just adding up the bytes

$$h(k) = \sum_{i=0}^{|k|} b_i.$$

An implementation of this approach in C is shown in Listing 2.1.

To achieve a better distribution, a hash function often includes a mixing step. Instead of addition, the key's bytes can also be combined by using the exclusive-or operator.

```c
uint32_t additive(char* key, uint32_t len)
{
    uint32_t hash, i;
    for (hash = i = 0; i < len; i++)
        hash += key[i];
    return hash;
}
```

Listing 2.1: Additive hashing, adapted from [Jen97].

```c
uint32_t rotating(char* key, uint32_t len)
{
    uint32_t hash, i;
    for (hash = len, i = 0; i < len; i++)
        hash = (hash << 4) ^ (hash >> 28) ^ key[i];
    return hash;
}
```

Listing 2.2: Rotation hashing, adapted from [Jen97].

A simple example making use of both mixing and exclusive-or combining is shown in Listing 2.2. There, the value of the variable `hash` is rotated to the left by four in every step to mix the data. Rotation by four means that the bits of the value get shifted to the left by four and the overflowing upper four bits are placed in the now empty lower four bits. The function adheres to the common hash function pattern shown in Listing 2.3.

Hash functions like the ones shown in Listings 2.1 to 2.2 output a fixed amount of bits, this behavior is called fixed-length hashing.

```
initialize(internal state)
for each block in key:
    combine(internal state, block)
    mix(internal state)
return postprocess(internal state)
```

Listing 2.3: Common hash function pattern, adapted from [Jen97].

2.7. Neural Networks

Artificial neural networks, often just called neural networks, are a computation concept inspired by the biological neural networks found in the brains of humans and other animals. In biological neural networks, neurons are connected by synapses which are used to transmit signals. Those connections are formed by learning processes during early brain development but continue to adapt more during a human's or animal's lifetime. Biological neural networks are able to perform tasks that are hard even for powerful computers in a fast and efficient manner. This includes tasks like image and pattern recognition (seeing), natural language procession (hearing and speaking) and predictions of various kind (where a thrown ball hits, the relative speed of another object etc.). Neural networks try to model this behavior by using artificial neurons.

Modern neural networks even achieve better-than-human scores on image recognition tasks [He+15].

A neural network is made up of several layers of those artificial neurons. This will be explained in Section 2.7.3. Beforehand, artificial neurons are described further.

2.7.1. Artificial Neurons

An artificial neuron is an information-processing unit that computes one output value given several input values.

Figure 4: artificial neuron

An artificial neuron k has a weight w_{ki} for each of its m inputs x_i, the weights and inputs are multiplied and then combined in a sum

$$u_k = \sum_{i=1}^{m} w_{ki} x_i. \tag{2.28}$$

Additionally, a bias b_k is added and the sum is put through an activation function σ. With this the output y_k of a neuron k is

$$y_k = \sigma(u_k + b_k). \tag{2.29}$$

Therefore, an artificial neuron is a function

$$f \colon X \to Y \tag{2.30}$$

with input space X and output space Y. Common choices for X and Y are $[-1,1]^n$ or $[0,1]^n$ where, for X, n is the number of inputs and $n = 1$ for Y. The reasons for this are related to the training and will be explained in Section 2.7.4.

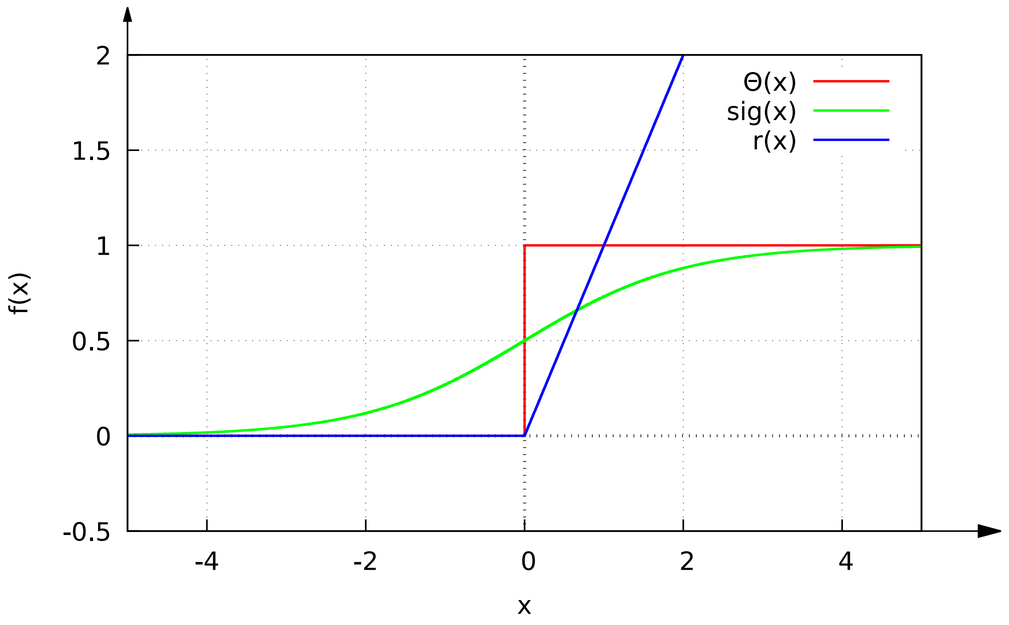


Figure 2.7.: Θ: Heaviside, sig: Sigmoid, r: ReLU

2.7.2. Activation Functions

Activation functions act as thresholds for the neurons. The intention behind them is that a neuron's inputs have to add up to a certain amount before its output is activated, hence the name. In this section, several activation functions are introduced.

Heaviside Function

The Heaviside function is a step function, named after Oliver Heaviside. Originally developed for operational calculus it has found widespread application in physics and other disciplines as a convenient way to model on/off relationships.

The Heaviside function $\Theta\colon \mathbb{R} \to \{0, 1\}$ is defined as

$$\Theta(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases} \tag{2.31}$$

This activation function was used in early neural networks, first introduced in 1943 [MP43]. However, in modern neural networks, it is not used as it renders the network unable to be trained by gradient descent as described in Section 2.7.4. The reason for this is that the derivative of Θ is not defined for $x = 0$ and is zero everywhere else, which causes the gradient to vanish.

Sigmoid Functions

A sigmoid function is a function $\mathrm{sig}\colon \mathbb{R} \to \mathcal{I}$ where either $\mathcal{I} = [0, 1]$ or $\mathcal{I} = [-1, 1]$ depending on the definition. In neural networks, the commonly found sigmoid functions are logistic functions and the hyperbolic tangent.

The standard logistic function is

$$\mathrm{sig}(x) = \frac{1}{1 + \exp(-x)}. \tag{2.32}$$

A visualization of this function is shown in Figure 2.7. The logistic function can be made steeper by introducing a factor α, so that

$$\mathrm{sig}_\alpha(x) = \frac{1}{1 + \exp(-\alpha x)}. \tag{2.33}$$

This modification can influence the learning speed immensely [HM95].

Interestingly the hyperbolic tangent is a scaled and offset version of the standard logistic function.

$$\tanh = 2\,\mathrm{sig}(2x) - 1 \tag{2.34}$$

Both the standard logistic function and the hyperbolic tangent are still used in neural network examples and teaching. However, due to their characteristic of being almost completely flat for input values outside of $(-4, 4)$, neural networks using them also tend to suffer from the vanishing gradient problem, especially in deeper networks. The vanishing gradient problem will be explained further in Section 2.7.4.

Rectified Linear Unit

Due to the limitations of logistic functions a different function was needed to efficiently train deep and very deep neural networks. This function turned out to be the ramp function $r\colon \mathbb{R} \to [0, \infty)$

$$r(x) = \max(0, x) \tag{2.35}$$

In the beginning, a node utilizing this activation function was called a Rectified Linear Unit (ReLU). Over time this name was also used to refer to the function itself. The derivative of r is the Heaviside function Θ. This helps with avoiding the vanishing gradient problem. Also, the ReLU is closer to the activation of real biological neurons and lets networks train faster than the hyperbolic tangent or the other logistic functions [GBB11] [KSH17].

State of the art image classification networks employ enhanced variants of ReLU, known as leaky ReLU (LReLU) and parametric ReLU (PReLU). These variants introduce an additional parameter $0 < a \leq 1$ to further reduce issues with the vanishing gradient problem.

$$r_a(x) = \max(x, ax). \tag{2.36}$$

In LReLU activation, this parameter is fixed while in PReLU activations it is trainable. Current image classification networks using PReLU achieve higher-than-human performance on the ImageNet 2012 dataset [He+15].

2.7.3. Feed-Forward Networks

When several neurons are connected they form a directed graph called a neural network. Those networks can be constructed to have layers. A layer is a set of neurons that are connected to two other sets (layers) of neurons for in and output, while not having any connections to each other. When the resulting graph has no circles the network is called a feed-forward network. When each neuron in a layer is connected to every node in the following and the previous layer it is called a dense layer. Layers that are between the in and the output layer and, thus, have no connections to the outside, are called hidden layers. Figure 2.8 shows a feed-forward network with four input nodes, five nodes arranged in one hidden dense layer and one output node.

To calculate the output of a dense layer, linear algebra is essential. Let n be the number of inputs (nodes in the previous layer) and m the number of outputs (nodes in the current layer) of the layer. Additionally, let $\mathbf{x}$ be the input vector composed of the inputs x_i. Furthermore, let $\mathbf{b}$ be the bias vector composed of the nodes biases b_j. Every of the m nodes has a weight vector $\mathbf{w_j}$, with $\mathbf{w_j} = (w_{j1}, \dots, w_{jn})$. The weight vectors form the weight matrix $W = (\mathbf{w_1}, \dots, \mathbf{w_m})^t$. Thereby, allowing to express the output $\mathbf{y}$ as

$$\mathbf{z} = W\mathbf{x} + \mathbf{b} \tag{2.37}$$
$$\mathbf{y} = \sigma(\mathbf{z}) \tag{2.38}$$

where σ is the component-wise activation function. This simple matrix-vector product allows fast computation as it is a common operation for which highly optimized libraries exist. It also allows for easy parallelization and benefits from processor instruction set extensions like *single instruction, multiple data* (SIMD) and *fused multiply-add* (FMA). SIMD allows doing certain floating-point operations like multiplication and addition on a vector of multiple values at once. FMA fuses a multiplication followed by an addition into a single operation. Obviously, both cases are given in a matrix-vector multiplication as in Equation (2.37).

The calculation of a whole neural network's output on a dataset is called *inference*. For a feed-forward network with n inputs and one hidden layer containing m nodes the inference time scales with $\mathcal{O}(nm)$.

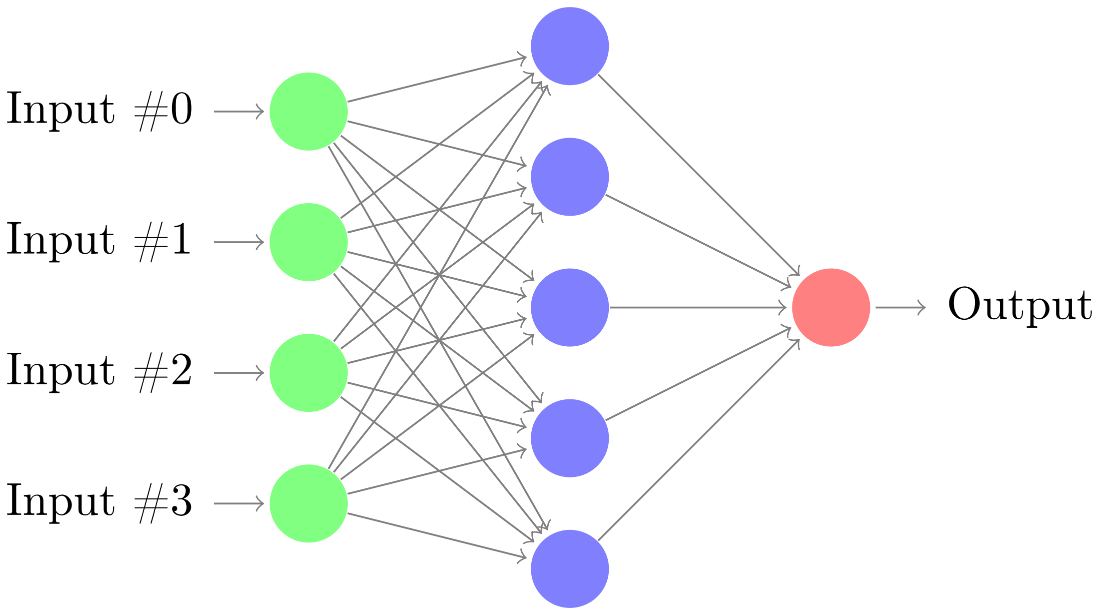


Figure 2.8.: Simple Feed-Forward Network

2.7.4. Training

Like their biological counterparts, neural networks need to be trained to learn what the right response for their input is. This is commonly done by supervised learning. In supervised learning, the network is fed with training data consisting of training inputs and their corresponding correct outputs. During training, the network's weights and biases are modified so that the network's output comes as close as possible to the correct outputs. This is done by first defining a loss function for the network and then minimizing it.

A common loss function is the L_2 loss

$$L_2(\mathbf{x}) = (F(\mathbf{x}) - y)^2 \tag{2.39}$$

where F is the network's output function and y the correct output for the training input $\mathbf{x}$. The L_2 loss is often chosen as it grows quadratically, penalizing large errors more than smaller ones. The average loss of the network is

$$L = \frac{1}{n}\sum_{i=1}^{n} L_2(\mathbf{x_i}) \tag{2.40}$$

This function is minimized by backpropagation and gradient descent explained in Section 2.7.4.

Gradient Descent

The training's goal is to minimize the loss function, more precisely to find a local minimum of it. Normally, in calculus, a function's extrema are found by finding a zero of its derivative. To identify if it is a minimum the second derivative of the function is evaluated at the found variable if it is positive the found point is a minimum. For functions of one variable, this is straightforward. However, the calculation already becomes much more complex for a function f of two variables x and y. There, instead of a simple derivative, the gradient $\boldsymbol{\nabla} f(x, y)$ is needed. The gradient of a function is the vector containing all the functions partial derivatives, in the case of the previously mentioned function f it is

$$\boldsymbol{\nabla} f = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix}. \tag{2.41}$$

After values for x and y are found so that $\boldsymbol{\nabla} f = 0$ the second derivative is given by the Hessian matrix

$$\mathrm{H}(f) = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} \end{pmatrix}. \tag{2.42}$$

If $\mathrm{H}(f)$ is positive semi-definite, the found point is a local minimum of f.

However, the loss function of a neural network depends on hundreds to billions of variables in a non-trivial way. Thus, this approach will not work for most neural networks. Hence, an alternative way to minimize the loss function is needed. A common approach to do this is the so-called gradient descent.

Let $L(v_1, \ldots, v_n)$ be the loss function. Then a change ΔL is related to changes Δv_i in the arguments of L. This relationship is approximately given by

$$\Delta L = \sum_{i=1}^{n} \frac{\partial L}{\partial v_i} \Delta v_i. \tag{2.43}$$

This can also be written as

$$\Delta L = \langle \boldsymbol{\nabla} L, \Delta \mathbf{v} \rangle \tag{2.44}$$

where $\langle \bullet, \bullet \rangle$ denotes the Euclidean scalar product and $\mathbf{v} = (v_1, \ldots, v_n)^t$. Now it is possible to chose

$$\Delta \mathbf{v} = -\eta \boldsymbol{\nabla} L \tag{2.45}$$

so that

$$\Delta L = -\eta \langle \boldsymbol{\nabla} L, \boldsymbol{\nabla} L \rangle = -\eta \|\boldsymbol{\nabla} L\|^2 \tag{2.46}$$

where η is a small positive number called the *learning rate*. ΔL is always negative as the norm of $\boldsymbol{\nabla} L$ is always positive. Thereby, a way of decreasing L until $\boldsymbol{\nabla} L = 0$ which indicates a found local minimum is given.

The gradient $\boldsymbol{\nabla} L$ can be calculated by repeated execution of the chain rule. The first step is shown in Equation (2.47). However, the further steps as well as the application of the changes to the weights and biases go beyond the scope of this thesis. In particular, Equation (2.47) causes every input $\mathbf{x}$ that produces an output $\mathbf{z}$ where the derivative $\sigma'(\mathbf{z}) = 0$ to be "stuck" as it does not actively influence the network's weights anymore.

$$\boldsymbol{\nabla} L = \boldsymbol{\nabla} \sigma(\mathbf{z}) = \sigma'(\mathbf{z}) \cdot \boldsymbol{\nabla} \mathbf{z} \tag{2.47}$$

Edge Case: Zero Hidden Layer Networks

When a neural network that uses the L_2 loss and has no hidden layers is used the training can be simplified immensely. Ignoring the activation function, the output function of such a network is

$$f(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle + b \tag{2.48}$$

where $\mathbf{w}$ is the weight vector and b is the bias. Obviously, this is equivalent to the equation

$$f(\mathbf{x}') = \langle \mathbf{w}', \mathbf{x}' \rangle \tag{2.49}$$

where $\mathbf{x}' = (\mathbf{x}, 1)$ and $\mathbf{w}' = (\mathbf{w}, b)$. Let now $X = (\mathbf{x_1}, \ldots, \mathbf{x_n})^t$ and $\mathbf{y} = (y_1, \ldots, y_n)$ be all the input values and the output values the networks is supposed to be trained on. Given the L_2 loss this is equivalent to solving

$$\min_{\mathbf{w}'} \|X\mathbf{w}' - \mathbf{y}\|_2^2. \tag{2.50}$$

Thus, the goal is to find the least squares solution $\mathbf{w}'$ for the equation

$$X\mathbf{w}' = \mathbf{y}. \tag{2.51}$$

Using the Gaussian normal equation, Equation (2.51) becomes

$$X^t X\mathbf{w}' = X^t\mathbf{y}. \tag{2.52}$$

This equation always has a solution and the solution is unambiguous if X is of full rank. If X is rank deficient the solution can be found with the pseudoinverse [DH08].

Summary

This chapter provided the essential knowledge of hash functions and neural networks necessary to design and evaluate an index model. First, the characteristics of file system metadata was explained, including file paths. Then parallel file systems were introduced and it was highlighted that they separate between the handling of data and metadata for performance reasons. The flexible storage framework JULEA and its usage of key-value stores was mentioned. The simple `put, get, delete` *interface of key-value stores was elaborated. Then, hash tables were explained in detail with a focus on the impact of collisions on their performance. Afterwards, regular hash functions were introduced and the source of their high collision ratio of 36% was discussed. Finally, neural networks and their training was explained. In particular, the troubles arising with vanishing gradients were highlighted.*

3. Design

This chapter provides insights on the approach taken to compare learned index structures to regular hash functions. First, the key-value store HTableDB and its components are introduced. Afterwards, the index model structure is explained further. Finally, the design decisions influencing HTableDB and its components are elaborated.

3.1. Approach

The goal of this thesis is to evaluate the performance of learned index structures, similar to the ones used in Section 1.2, in the context of high-performance storage applications. As storage frameworks like JULEA, mentioned in Section 2.3, use key-value stores to store metadata, the choice was made to test the learned index in a similar fashion. For this, an in-memory key-value store was implemented to enable the comparison of different index models and hash functions. The underlying index structure used is a hash table, as the structure enables the storage of new entries in contrast to the tree-like approach from "The Case for Learned Index Structures" [Kra+18]. Therefore, the implementation of a key-value store becomes possible. The key-value store and the index model were designed to use strings, e.g., file paths as keys. Further details on the used index models are given in Section 3.2. The developed key-value store is named HTableDB and is detailed further in Section 3.3.1.

HTableDB was designed to be compatible to JULEA's key-value store backend. In particular, HTableDB was designed to wrap around standard hash table implementations to take care of operation batching and to handle and provide multiple hash functions and index models to determine an entry's position. To facilitate an easy exchange of the hash table implementation and the used index model a layered modular approach was chosen. Figure 3.1 shows the three separate components: HTableDB, HTable and Indexnet. An application interacts only with HTableDB without any knowledge of the underlying components. HTable is the hash table implementation used to store the key-value pairs. The interaction with the hash table is done through a clean interface as the hash tables internal state is private. This enables an easy exchange of the hash table implementation in the future. Indexnet handles the index model, its internal state is private as well. Indexnet provides a simple index model based on a two-stage feed-forward network. It could be easily replaced by other index models for further tests in the future.

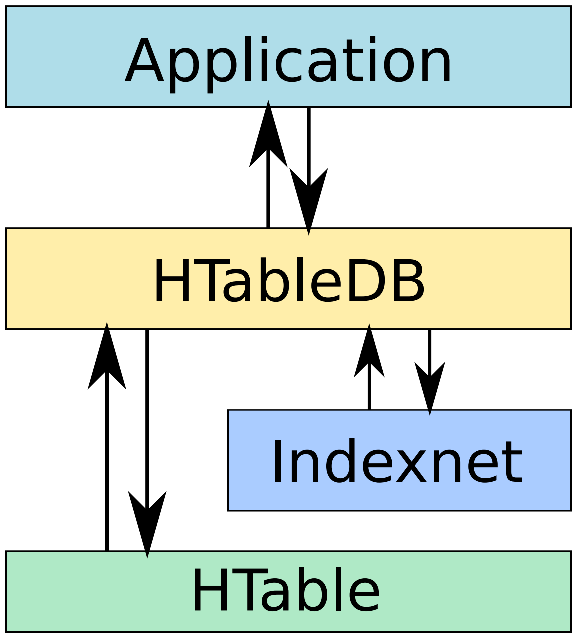


Figure 3.1.: Interaction of an application with HTableDB and its components

3.2. Index Models

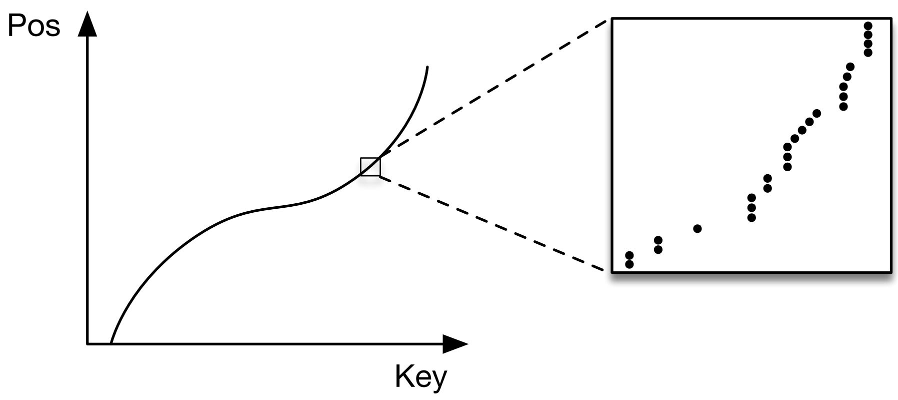


Figure 3.2.: Key indexes as cumulative distribution function.
Taken from [Kra+18]

An index model is a machine learning model used to calculate an index position from a key. This is done by scaling the model's output with the number of possible indexes. The basic approach is to view the key indexes as a cumulative distribution function as seen in Figure 3.2 and to label them accordingly. For integer keys this is straightforward. The keys are first sorted in ascending order. Afterwards, each of the n keys k_i is assigned a label $\ell(k_i) = {}^{i}\!/_{n}$. For strings, the labeling can be done using a different sorting approach like ordering by string numerical value.

The index models used in this thesis are feed-forward networks ordered in two stages. To process integer keys, such a network only needs one input node. To input strings, a common approach is to use one input node per character in the string. As the number of input nodes is constant, the strings have to be of a limited length. The ASCII values of the characters are divided by 127 to scale them to values in $[0, 1]$ before putting them into the network. If a string is shorter than the number of input nodes the remaining inputs are set to zero. The number of input nodes chosen depends on the keys used.

The first stage of the index model is a single feed-forward network f_0 with zero or one hidden layer. The second stage consists of m feed-forward networks $f_1^{(j)}$ with zero hidden layers. To calculate the index of a key x, first, the output $f_0(x)$ of the first network is computed. Then, the index of the subnet to be used is calculated. Afterwards, the output of the subnet is computed and scaled with the total number of keys to calculate the final index. In short

$$F(x) = f_1^{\lfloor m \cdot f_0(x) \rfloor}(x). \tag{3.1}$$

Thereby, the two-stage loss of key x is

$$L(x) = (F(x) - y)^2. \tag{3.2}$$

3.3. Components

3.3.1. HTableDB

HTableDB is the overlaying component and provides the external interface of the key-value store. It wraps around HTable and offers the user the choice between multiple hash functions and index models.

Currently, HTableDB offers the choice between using the trivial hash function shown in Equation (2.5), an adapted version of the Linux kernel's jhash [JK10], another hash function called fast-hash [Tan15] used in Apple's iOS kernel and GNU Hurd as well as position determination by Indexnet. Both jhash and fast-hash are in their function similar to the hash function shown in Listing 2.2. The differences are a more sophisticated mixing and that the data is read in chunks to increase the performance.

If one of the regular hash functions is chosen during the initialization, it is passed to HTable. The following operations are directly passed through from HTableDB to HTable which then uses the provided hash function with modulo reduction, as shown in Equation (2.2), to determine the position of the entries. This behavior is visualized in Figure 3.3.

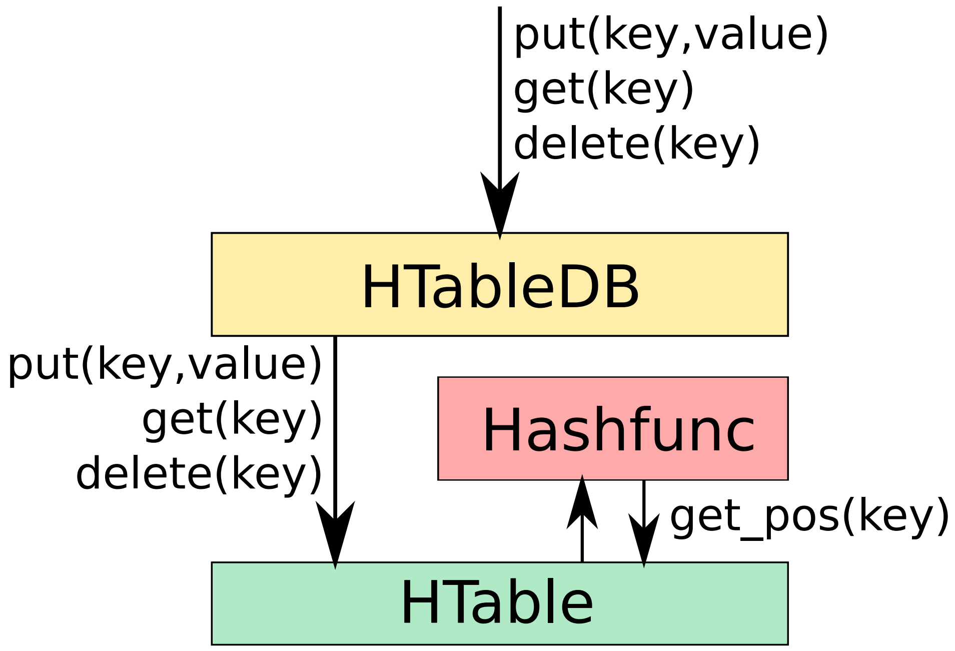


Figure 3.3.: HTableDB operations using regular hashing

If Indexnet is chosen, HTableDB takes care of Indexnet's initialization. When one of the HTableDB operations is called the entry position is determined by a call to Indexnet and then passed to HTable. This is shown in Figure 3.4.

3.3.2. HTable

HTable is a small hash table implementation using separate chaining with head entries as described in Section 2.5.1. Every bucket of the hash table stores three pointers, one to the key, one to the value and one to the next entry in the chain. This approach was chosen as it offers a good trade-off between memory demands of the bucket array and latency of the pointer traversal. The chains are made of singly linked lists.

Interface

The interface of HTable is shown in Listing 3.1. Similar to HTableDB, HTable is initialized with a function named `hashtable_init` which then returns a pointer to

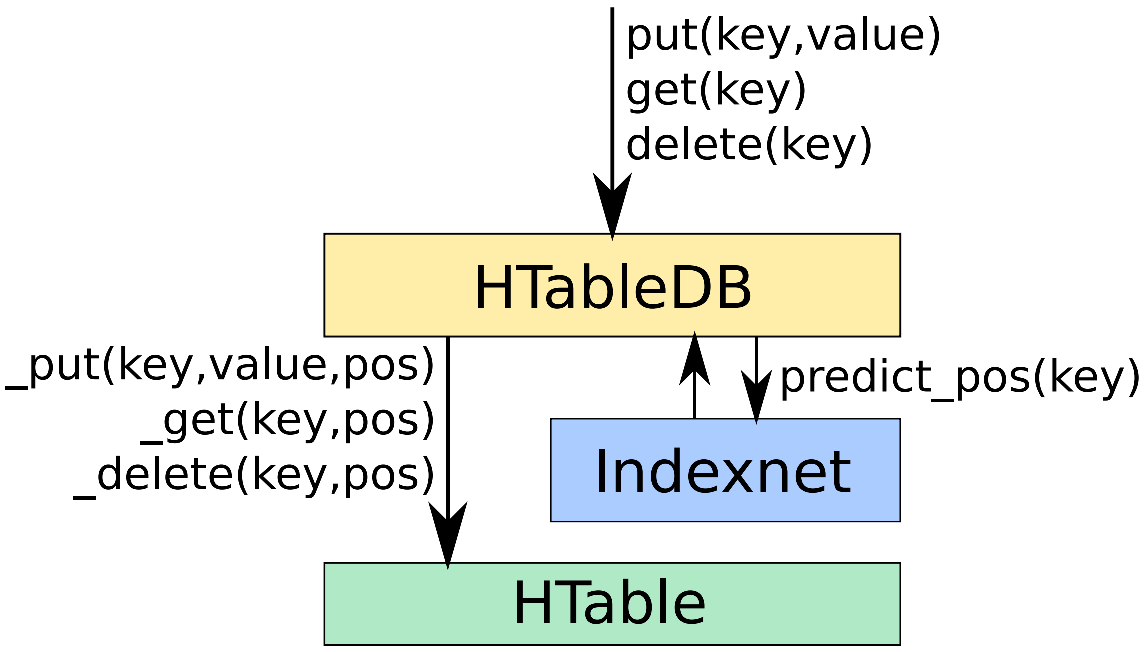


Figure 3.4.: HTableDB operations using Indexnet

a `hashtable` struct containing the state of the hash table HTable. This struct is needed by the remaining functions. During initialization, a hash function and a compare function have to be provided. When the hash table is no longer needed its memory has to be freed with `hashtable_free`. The hash table can either be used regularly with the provided hash function or with externally calculated positions. To store, retrieve or delete entries the functions `hashtable_put`, `hashtable_get`, and `hashtable_del` are used. When the position is determined externally, the functions have to be prefixed with an _ and the position has to be given as an argument. HTable has additional functions to get data about its size, the number of total collisions and the maximum chain length.

3.3.3. Indexnet

Following the index model approach described in Section 3.2 a two-stage model was chosen. The first stage consists of a feed-forward network with one hidden layer (see Section 2.7.3). The second stage consists of several feed-forward networks with zero hidden layers (linear regression). The number of epochs trained, the number of nodes in the hidden layer and the number of second stage networks was varied. The results of this variation will be presented in Chapter 5. The rectified linear unit (ReLU) described in Section 2.7.2 was used as activation function. Furthermore, the sigmoid activation function was tried. The number of input nodes was varied depending on the dataset used.

```c
struct hashtable* hashtable_init(uint64_t size, uint64_t
    ↪ (*hash_fn)(void*), bool (*cmp_fn)(void*, void*));

int hashtable_free(struct hashtable* ht);

int hashtable_put(struct hashtable* ht, void* key, void*
    ↪ value);

int hashtable_del(struct hashtable* ht, void* key);

void* hashtable_get(struct hashtable* ht, void* key);

void _hashtable_put(struct hashtable* ht, void* key, void*
    ↪ value, uint64_t pos);

int _hashtable_del(struct hashtable* ht, void* key,
    ↪ uint64_t pos);

void* _hashtable_get(struct hashtable* ht, void* key,
    ↪ uint64_t pos);

uint64_t hashtable_get_collisions(struct hashtable* ht);

uint64_t hashtable_get_max_chain_length(struct hashtable*
    ↪ ht);

uint64_t hashtable_get_size(struct hashtable* ht);
```

Listing 3.1: HTable Interface

```c
struct indexnet* indexnet_init(char* filename);

void indexnet_predict(struct indexnet* inet, const float*
    ↪ x, int batch_size, float* y);

void indexnet_free(struct indexnet* inet);

int indexnet_get_i_size(struct indexnet* inet);
```

Listing 3.2: Indexnet Interface

Interface

The interface of Indexnet is shown in Listing 3.2. Again, the internal state of Indexnet is saved in an opaque `struct indexnet`. A pointer to this struct is returned by the initialization function `indexnet_init` which takes the file name of a weight file as an argument. The structure of this file is explained in Section 4.3.1. The weight file contains the parameters and weight necessary to initialize the internal neural networks. If the instance of Indexnet is no longer needed it has to be freed by calling the `indexnet_free` function. To let users of Indexnet determine the right input size, the `indexnet_get_i_size` function exists. The remaining function `indexnet_predict` grants access to the main functionality, calculating the relative position in the bucket array for a given key. It takes a pointer to a float array `x` containing `batch_size` many keys of the input size. It also takes a pointer to another float array `y` of size `batch_size` which is used as the output.

Summary

This chapter provided insights into the design decisions taken and the overall approach. First, the key-value store HTableDB was introduced. It was developed to easily compare learned and unlearned index models. Then, the two-stage network approach was explained. It consists of a first stage made out of one feed-forward network which maps keys to a second layer of subnetworks. Afterwards, HTableDB was detailed further. The underlying components and their interfaces were introduced.

4. Implementation

This chapter provides further insights into the implementation details of HTableDB and its components. First, HTableDB and its interface are discussed. Then, the hash table used in HTable and the reasons for its implementation are explained. In the end, the separate training and inference of Indexnet is described.

4.1. HTableDB

As most file systems are written in the C programming language, the choice was made to use it for this component as well. C is a compiled, low-overhead programming language offering great portability as compilers exist for nearly any platform.

HTableDB's interface is shown in Listing 4.1. A HTableDB instance is initialized by the `HTableDB_init` function which returns a pointer to a `struct htabledb`. The initialization function takes two arguments `size` and `hash_type`. `size` is the number of buckets in the underlying hash table. `hash_type` sets the hash function used. It can be set to `TrivHash, JHash, FHash` or `IndexNet` to either use one of the regular hash functions trivial hash, jhash and fast-hash or to use the index model provided by Indexnet. To finalize the HTableDB instance the `HTableDB_fini` function is used. The three key-value store operations put, get and delete are implemented by the functions `HTableDB_put, HTableDB_get` and `HTableDB_delete`. These are implemented by wrapping around the HTable functions to handle the differences between the regular hash functions and the index models.

Additionally, the `HTableDB_print_debug` function was implemented. It prints several parameters to the console. The parameters are the hash table's total number of buckets, number of entries, number of collisions, collision ratio, max chain length and the size in bytes as well as the number of bytes wasted due to collisions. This function is used to print status information during benchmarks.

4.2. HTable

Like HTableDB, the implementation of HTable was done in the C programming language. The choice to implement a hash table from scratch was made after evaluating

```c
enum HASH_TYPE {TrivHash, JHash, FHash, IndexNet};

struct htabledb* HTableDB_init(uint64_t size, enum
    ↪ HASH_TYPE hash_type);

void HTableDB_fini(struct htabledb* htdb);

int HTableDB_put(struct htabledb*, void* key, void* value);

int HTableDB_delete(struct htabledb*, void* key);

void* HTableDB_get(struct htabledb*, void* key);

void HTableDB_print_debug(struct htabledb*);
```

Listing 4.1: HTableDB Interface

several other hash table implementations e.g. glibc's GHashTable. Most hash table implementations offered features unnecessary for the planned benchmarks like resizing. Additionally, those implementations were hard to modify to output additional information and allow direct placement of entries needed with Indexnet as lots of unknown code would have to be examined. The simple linked list implementation used for the chaining was also implemented from scratch. Currently, it is located inside of HTable.c, but in the future, it could be externalized to be used for HTableDB's batching.

4.3. Indexnet

4.3.1. Training

Initially, the full first stage training was implemented from scratch in C. However, the training loss never decreased. In retrospect, this could have had two reasons. The first one being that the paths were not sorted before being labeled. The second reason being that the initialization of the neural network's weights was always the same as it was done with the same seed and not done carefully enough. In particular, the random number generator used was `drand48` which is a linear congruential generator and known to produce pseudo-random numbers that do not pass some of the statistical tests random numbers should pass [LS07].

To rule out the training implementation of the first stage as a source of error, the well-tested Keras [Cho+15] Python library with TensorFlow [Mar+15] as backend was used. The weights of the second stage networks are calculated with the linear regression method described in Section 2.7.4 using the NumPy [Oli06] Python library. Afterwards, the weights and biases of the first and second stage networks are written to a binary file. As NumPy internally uses C arrays this process is straightforward and guarantees compatibility with C code executed on the same machine. This binary file has three parts: the first part is 24 bytes long and contains the number of inputs n, the number of nodes m in the hidden layer and the number of second stage networks ℓ as signed 64 bit integers. The second part is $(nm + 2m + 1) \cdot 4$ bytes long and contains the weights and the biases of the first stage's hidden layer and the weights and the bias of the first stage's output node as 32 bit floating point numbers. The third part is $\ell(n + 1) \cdot 4$ bytes long and contains the weights and the bias of the second stage networks, again as 32 bit floating point numbers.

4.3.2. Inference

The inference step is done by a C implementation made from scratch. Not only does this help to reduce the overhead induced by the TensorFlow and Python runtime environments, but also aids portability by reducing dependencies. Training and inference can be done on different computers as long as their word size and byte order (endianness) are the same. This constraint could be lifted by storing the network's data in a self-describing data format e.g. HDF5 [The18]. This was not yet implemented as it goes beyond the scope of this thesis.

As mentioned in Section 3.3.3, the inference can be done in batches. This is achieved by a generalization of the Equations (2.37) to (2.38). Instead of just one input vector $\mathbf{x}$, a number of n input vectors $\mathbf{x_j}$ is composed into an input matrix $X = (\mathbf{x_1}, \dots, \mathbf{x_n})$. Hence, the new equations are

$$Z = WX + B \tag{4.1}$$

$$Y = \sigma Z \tag{4.2}$$

where σ is the component-wise activation function. The matrix B is composed out of the layer's bias vector n times side by side.

$$B = (\mathbf{b}, \dots, \mathbf{b}). \tag{4.3}$$

However, the batching is not used at the moment as the batching functionality of HTableDB remains future work.

For the second stages, the inference step is not done in batches as it would require an expensive reordering of the input array. Thus, the inference is still the scalar product shown in Equation (2.48).

To implement the matrix-matrix multiplications and the scalar product the well-established *Basic Linear Algebra Subprograms* (BLAS) library was chosen. In spite of being written in Fortran, BLAS provides a C interface called CBLAS. CBLAS offers a well-defined interface which enables the code to be linked against several optimized implementations of the linear algebra operations like the Intel® Math Kernel Library (Intel® MKL) [Int18] or OpenBLAS [Xia+18]. In particular, the two functions used by Indexnet are `cblas_sgemm` and `cblas_sdsdot`. The `sgemm` function is used for *single-precision general matrix multiplication* as occurring in the first stage (see Equation (4.1)). The `sdsdot` function implements a single-precision scalar product with additional precision during the accumulating step and is needed for the second stages (see Equation (2.48)).

Summary

In this chapter, first, the HTableDB key-value interface with its `put, get, delete` *interface was presented. Then, the choice to implement a hash table from scratch for HTable was explained. This choice was taken as it enabled faster integration of additional functions for the gathering of performance statistics. Afterwards, the Keras, TensorFlow, Python stack of Indexnet's training was introduced. Finally, Indexnet's inference step and its batching was explained.*

5. Evaluation

In this chapter, the learned index models will be evaluated. First, the influence of a network's parameters on its loss will be analyzed. Then, the distribution of entries to subnets as well as the influence of the number of subnets on the loss will be studied. Finally, an index model will be compared to the jhash and fast-hash hash functions.

Datasets

Several datasets were used during the evaluation. As a foundation, a dump of all the file paths of a file system, used for multiple scientific software workloads in a small cluster, was acquired via `find /`. Afterwards, the paths longer than 256 characters were removed and the order of the paths was shuffled. Then, it was split in two with each dataset containing 2180000 file paths. Finally, those datasets were sorted with `sort -n`. These datasets will be called set A and set B. Additionally, a third dataset was created containing a subset of the initial file paths. That subset contains the unpacked ImageNet dataset [Rus+15] and will be called dataset I. Dataset I has 1878718 entries.

5.1. Optimization of First Stage Loss

The first step taken was to optimize the training parameters of Indexnet's first stage. For the first stage with zero hidden layers, this is trivial as there are no training parameters. This is due to the closed form solution explained in Section 2.7.4. To find the optimal model parameters for the first stage with one hidden layer, the grid search approach was used. Grid search is the evaluation of a model for every parameter p in a parameter space $\mathcal{P}$. The parameter space searched was

$$\mathcal{P} = \{1, 2, 4, 8\} \times \{1, 2, 4, \dots, 512, 1024, 2048\} \tag{5.1}$$

with $\mathcal{P} \ni p = (i, j)$ where i is the number of epochs the network was trained for and j the number of nodes in the hidden layer. For every parameter p the network was trained five times on dataset A. Results were the network did not converge, i.e. where the loss did not decrease, were removed. Afterwards, the arithmetic mean of

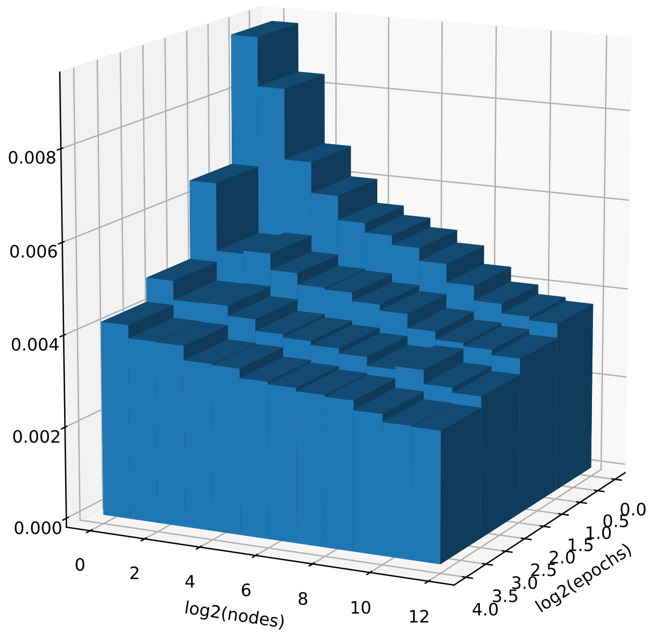


Figure 5.1.: Loss for a given node/epoch combination (lower is better).

the results was calculated and visualized in Figure 5.1. Although the training time was still in the range of tens of minutes for the tested number of epochs, the number of epochs for further measurements was set to four as it was a good trade-off between training time and loss minimization. In practice, the network could be trained for more epochs as the training is only done once and the number of epochs has no influence on the time needed for inference. However, the training becomes slower over time while the risk of overfitting increases. The number of nodes in the hidden layer was set to 16 as more nodes only marginally lowered the loss while directly increasing the inference time. The inference time of a feed-forward network with one hidden layer scales with $\mathcal{O}(i\ell)$ where i is the number of inputs and ℓ the number of nodes in the hidden layer. For this reasons, $p = (4, 16)$ was chosen for some of the further measurements.

The average first-stage loss for a network with 16 nodes in the hidden layer trained for four epochs was approximately `0.00387`. For 2048 nodes and eight epochs the average loss was approximately `0.00282`. However, for a simple first-stage with zero hidden layers the loss was already `0.00329`, making networks like the first one potentially irrelevant in practice.

5.2. Distribution of Entries to Subnets

Another important aspect is the distribution of keys to the second-stage networks also called subnets. Ideally, every subnet should get an equal amount of entries as the subnets partition the bucket array into equally sized chunks. Thus, a subnet getting a number of entries higher than the total number of entries divided by the number of subnets inevitably leads to collisions. The labels for the training data are chosen to be equally partitioned, therefore, minimization of the loss should lead to equally distributed entries per subnet. This was again tested on dataset A.

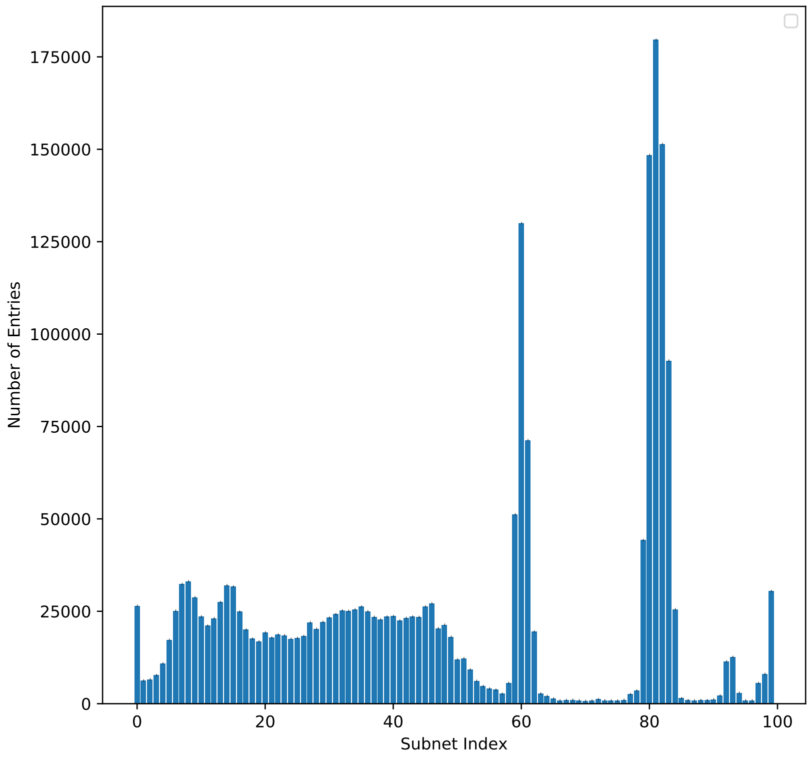


Figure 5.2.: Number of entries per subnet for 100 subnets and a first stage with 16 nodes in the hidden layer, trained for 4 epochs. First-stage loss: `0.00387`, total loss: `0.000991`.

Figure 5.2 shows the distribution of entries to 100 subnets with a first-stage network with one hidden layer containing 16 nodes trained for four epochs. As the size of the dataset was 2180000 entries, subnets with more than $^{2180000}/_{100} = 21800$ entries inevitably cause collisions. The figure shows that the first 50 subnets got a number of entries that can be considered close to optimal. Only a couple subnets after the first got a number of entries lower than optimal. The subnets around the indexes 60 and 80 got numbers of entries exceeding the optimal value by a factor up to 6, while the subnets between them are nearly empty. The remaining subnets are also close to empty. The last and the first subnet are standing out as they catch all the entries where the networks output before the last activation function is negative respectively greater than one. In the following, the subnet distribution for several different first-stage parameters is discussed.

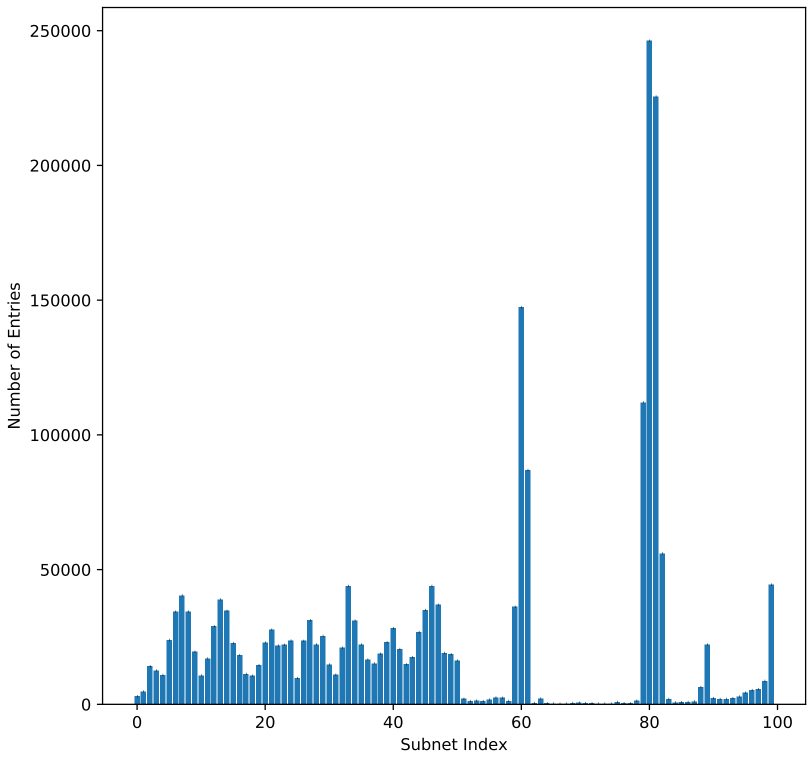


Figure 5.3.: Number of entries per subnet for 100 subnets and a first stage with zero hidden layers. First-stage loss: `0.00329`, total loss: `0.000883`.

Figure 5.3 shows the subnet distribution for a zero hidden layers network and 100 subnets. The overall distribution is similar to the one in Figure 5.2, just not as smooth for the first half of the subnets. However, the loss of the first stage, as well as the total loss over both stages, was lower. Additionally, the peaks at 60 and 80 are sharper and higher than before. Also, the peaks in the first half are over 25000 entries per subnet and will cause additional collisions. In conclusion, this net should perform worse when used for indexing even though its loss is lower.

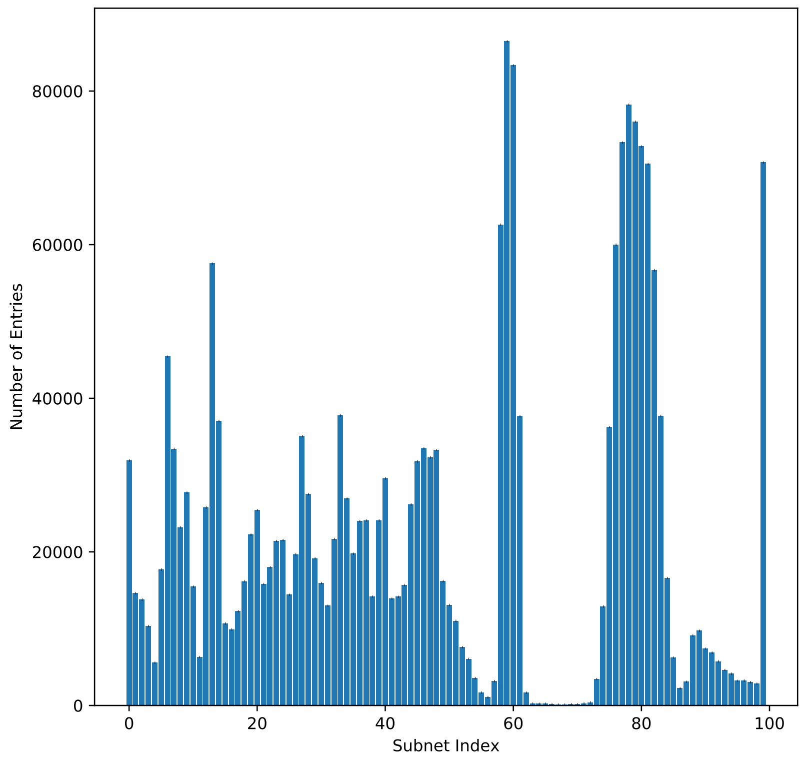


Figure 5.4.: Number of entries per subnet for 100 subnets and a first stage with 16 nodes in the hidden layer, trained for 32 epochs. First-stage loss: `0.00221`, total loss: `0.000371`.

In an attempt to smooth the distribution seen in Figure 5.2 further, the number of epochs was increased to 32. The result of this is shown in Figure 5.4. The peaks around 60 and 80 got broader and lower as expected. However, the previously smooth first half of the subnet distribution has become rougher, with lots of subnets having either half or twice as many entries as optimal. The subnets between the two main peaks remain empty. The last subnet has over 70000 entries, indicating that the network tries to output a lot of values greater than one. This is problematic as the gradient of the ReLU activation function is zero for those values, thus, they are not actively influencing the gradient descent anymore. Therefore, the weights influencing them the most are "stuck". The network can only recover from this problem if there are still values for which the activation function's gradient is not zero connected to those weights. Again, the loss has decreased compared to the previous networks. The number of epochs was further increased to 64 and 128 with similar results. Again, the distribution became less smooth while the loss decreased. This behavior can be seen in Figures A.1 to A.3.

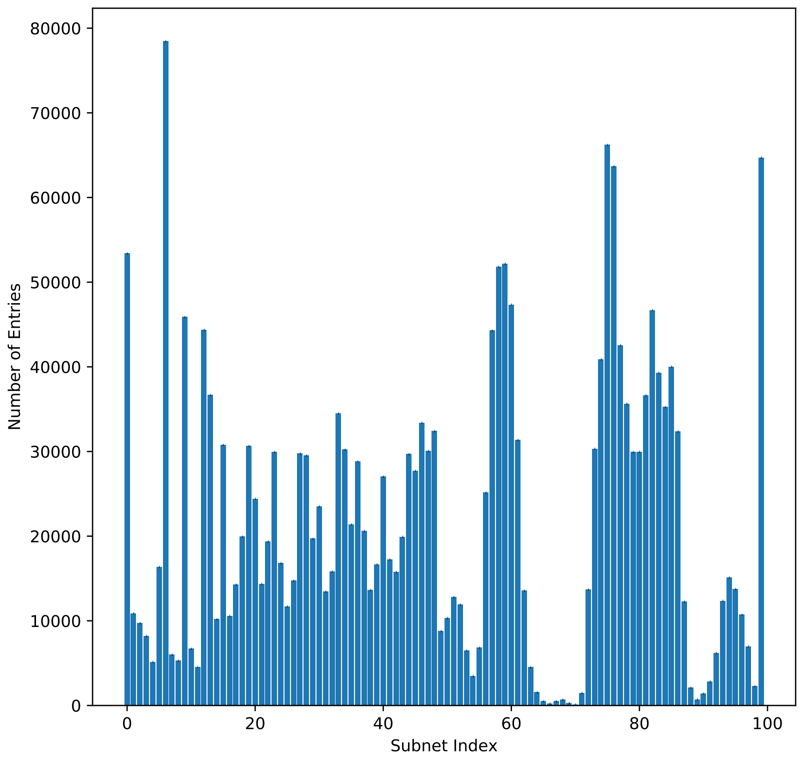


Figure 5.5.: Number of entries per subnet for 100 subnets and a first stage with 64 nodes in the hidden layer, trained for 64 epochs. First-stage loss: `0.00120`, total loss: `0.000187`.

As the grid search results suggested, more nodes in the hidden layer trained for more epochs further decrease the loss. The numbers of epochs and hidden layer nodes were set to 64. The results are visualized in Figure 5.5. The first-stage loss, as well as the total loss, have both approximately halved from the previous network. Additionally, the peaks at 60 and 80 have become less pronounced. However, the distribution of entries in the first half became worse having single spikes up to 80000 entries per subnet. Like before, the first and the last subnet have a lot of entries indicating that the network architecture is not optimal.

After doubling the number of hidden layer nodes and number of epochs to 128, the loss stays approximately the same as before. The inequalities in the distribution of the first half of the subnets became more extreme. This behavior is shown in Figure 5.6. When the number of nodes in the hidden layer is increased further to 256, the overall distribution stays the same. However, the number of entries in the last subnet has increased further. This can be seen in Figure A.4.

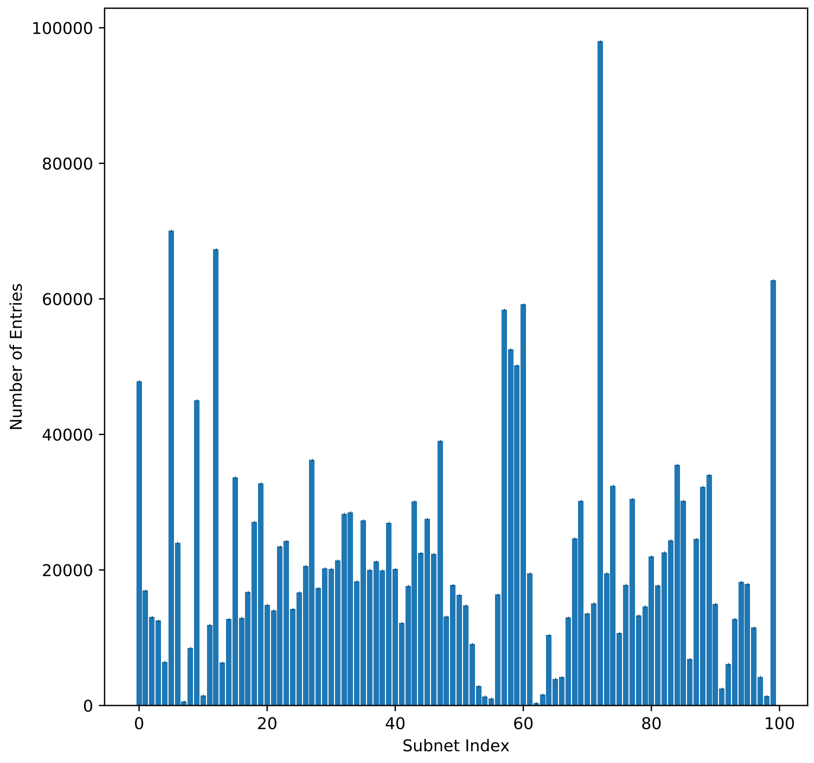


Figure 5.6.: Number of entries per subnet for 100 subnets and a first stage with 128 nodes in the hidden layer, trained for 128 epochs. First-stage loss: `0.000673`, total loss: `6.455e-5`.

In conclusion, the measurements in this section suggest that the chosen first-stage network architecture in combination with the equal partitioning of the bucket array by the subnets is not ideal. The first-stage is unable to distribute the tested entries equally, independent of the choice between a zero and a one hidden layer network. This also occurs when dataset I is used as shown in Figures A.5 to A.11. To improve further, either a different first-stage architecture is needed or the way the subnets partition the array has to be more dynamic, attributing for the number of entries mapped to a subnet. Additional approaches to address this problem will be discussed in Chapter 6.

5.3. Number of Subnets

The impact of the number of subnets on the total loss over both stages was evaluated as well. For this, the number of hidden layer nodes was set to 16 and the network was trained for four epochs. Afterwards, the number of subnets was varied and the loss was calculated. The results are shown in Figure 5.7. After 1000 subnets, which is equal to one subnet for every 21800 entries, the loss only marginally decreased, suggesting that more subnets are unnecessary. The data that Figure 5.7 is based on can be seen in Table A.1. The initial step-like decrease of the loss likely stems from peaks as the ones discussed in Section 5.1 being distributed to multiple subnets.

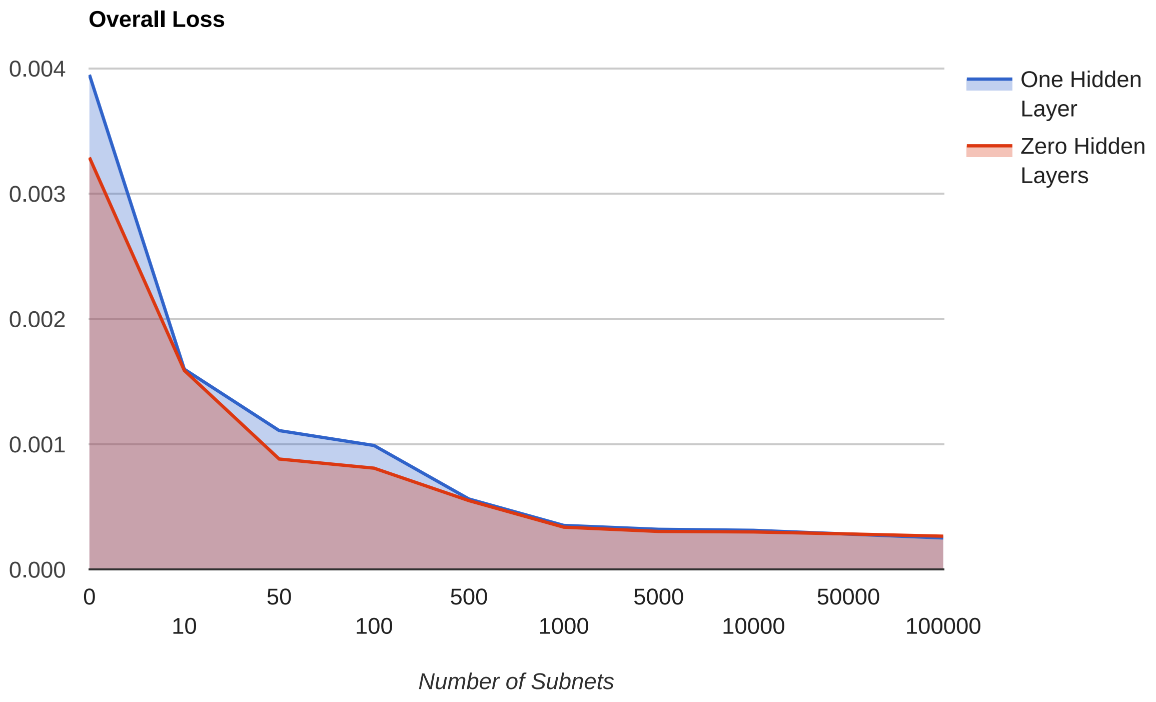


Figure 5.7.: Overall loss by number of subnets

5.4. Comparison to Hash Functions

To compare the learned index models to regular hash functions HTableDB was used. Dataset B was chosen for the comparison to test how well the neural network generalized to previously unseen data. Additionally, dataset I was used to evaluate the performance of learned indexes on homogeneous data. The benchmark used consisted of three parts. First, every entry of the dataset was put into the HTableDB. Afterwards, a get request was made for every entry. Finally, all the entries were deleted. In total, the benchmark causes $3n$ lookups for n entries in a dataset.

Method	Time	Collision Rate	Max Chain Length	Bytes Wasted
Fast-Hash	0m2,645s	0.368	9	18M
JHash	0m2,765s	0.367	10	18M
Indexnet	33m24,249s	0.789	210507	39M

Table 5.1.: Benchmark results for dataset B

Test Environment

Hardware used:
Intel® Core™ i5-4210U processor
2×4 GiB 1600MHz DDR3L

5.4.1. Results for Dataset B

The results for dataset B are shown in Table 5.1. The index model used in Indexnet had one hidden layer with 16 nodes trained on dataset A for four epochs. It was the most promising candidate after analysis of the results in Section 5.1. However, its performance was not sufficient. Apparently, large numbers of the paths from the input data were mapped to the same bucket arrays, not only causing a high collision ratio but also really long chains. Further analysis with `gprof` showed that a significant part of the execution time was spent traversing the chains. The longest chain was 210507 entries long, it alone caused

$$\frac{210507 \cdot 210508}{2} \cdot 3 = 66470111334 \qquad (5.2)$$

linked list hops. If all of those hops were cache misses as explained in Section 2.5.1 the whole chain would have added over one hour to the benchmark time. The jhash and fast-hash hash functions performed as expected. In particular, they achieved a collision ratio of ≈ 0.367 as calculated in Section 2.6.1.

5.4.2. Results for Dataset I

The results for dataset I are shown in Table 5.2. The index model used had one hidden layer with 16 nodes trained on dataset I for four epochs. The collision rate of the index model was 0.432 which is worse than that of the hash functions. However, the overall performance was better than for dataset B and further improvements might be possible if the large number of entries mapped to the last slot can be reduced. The hash functions performed as expected.

Method	**Time**	**Collision Rate**	**Max Chain Length**	**Bytes Wasted**
Fast-Hash	0m1,728s	0.367	8	16M
JHash	0m1,700s	0.367	8	16M
Indexnet	1m17,988s	0.432	61557	18M

Table 5.2.: Benchmark results for dataset I

Summary

In this chapter, various aspects of an index model as described in Chapter 3 were evaluated. Evaluation of the loss reduction and subnet distribution showed that minimization of the average mean squared error does not necessarily lead to a good index model in the short term. Furthermore, problems with the used network architecture were found, in particular, the vanishing of the gradient for a significant portion (> 10%) of the test data. For general datasets containing very different paths, the tested networks were unable to distinguish similar paths sufficiently. However, for datasets containing similar paths, the tested networks performed better.

6. Summary, Conclusion and Future Work

This chapter summarizes and concludes the thesis. Additionally, this chapter discusses future work regarding learned indexes and the developed components.

6.1. Summary

In Chapter 1, it was stated that exploitation of data's inner structure by learned index models could lead to better performing index structures. As a use case hash tables were introduced and the goal to compare an index model's performance against regular hash functions was defined. Chapter 2 provided the essential knowledge to achieve this goal. It introduced file system metadata and file paths, the separate handling of data and metadata by parallel file systems and the novel approach of using object and key-value stores taken by modern storage frameworks like JULEA. Furthermore, hash tables and hash functions were explained with a focus on their performance degradation due to collisions. In Chapter 3, the decision to implement a key-value store was reasoned. The modular design for future extensibility of HTableDB was explained and the index model architecture was detailed. The index model used consists of two stages of feed-forward networks. The first stage consists of one network that maps keys to one of the subnetworks for index calculation. Chapter 4 further detailed the implementation of HTableDB and its components. It elaborated the Keras, TensorFlow, Python stack used for training and the BLAS interface used for inference. Chapter 5 evaluated the learned index models used. It was found that minimizing the average mean squared error does not necessarily lead to a good index model in the short term. Furthermore, it was found that with the used architecture about 10% of the input keys cause the gradient to vanish and therefore inevitably cause a collision. Additionally, these values are "stuck" and, thus, further training does not resolve the issue. For datasets containing homogeneous paths the index model performed slightly worse than the hash functions.

6.2. Conclusion

Chapter 5 showed that simple feed-forward networks with one hidden layer do not perform sufficiently to be competitively used to index large amounts of non-similar file paths. However, it was shown that they already come close to the collision rates of commercial-grade hash functions when used for file paths that are more similar to each other. A potential cause for their weak performance with the datasets A and B introduced in Chapter 5 is that both datasets contain strongly clustered parts of dataset I. An additional weak point of the taken approach was the supervised training. Supervised training needs pre-defined labels which impose a strong structure on the data. If this structure is not similar to an internal “hidden” structure of the data the network is unable to learn a good distribution. The imposed structure was labeling by increasing numerical string value (`sort -n`), however, a different structure might be more optimal. A significant portion of the tested dataset’s entries was mapped to either the first or the last slot in the hash table, thus, indicating problems with the network’s architecture. If those problems can be resolved the index model has the potential to be a valid competitor to the regular hash functions when considering space efficiency.

6.3. Future Work

HTableDB

During the design of HTableDB and its components decisions were made to increase their extensibility. Batching is already supported by Indexnet and would lower the overall index calculation time by efficiently calculating the indexes for multiple keys at once. Additionally, the BLAS interface offers further performance gain by using a variant optimized more than the reference implementation. As HTableDB’s design was influenced by JULEA’s key-value backends the implementation of such a backend for HTableDB is feasible and offers a good way to further test learned indexes in real-life applications. After further modularization Indexnet could become a basis that supports various index model architectures. One of those modularization steps would be to support the HDF5 model and weight files as exported by Keras, drastically simplifying the testing of new index models.

Additional Index Models

Simple feed-forward networks showed various shortcomings when indexing file paths, however, the variety of machine learning models still offers different architectures to be tested. Two of them are *self-organizing maps* (SOM) and *one-dimensional convolutional neural networks* (1D-CNN). Self-organizing maps are trained unsupervised and thus are not influenced by suboptimal structures chosen by the user. Convolutional neural networks in their 2D and 3D variants are largely used in current image recognition networks. They have also been successfully used in natural language processing, analyzing sentences [CW08].

Summary

This chapter summarized and concluded this thesis. First, it recalled the overall structure of the thesis and summarized its chapters. Then, it was concluded that the tested index models do not yet perform sufficiently but show room for improvement. Finally, possible future work was discussed, including improvements to HTableDB as well as testing of machine learning models like self-organizing maps or convolutional neural networks.

Bibliography

[7CPU] *7-CPU Broadwell Test*. Last accessed: 24.08.2018. URL: `https://www.7-cpu.com/cpu/Broadwell.html`.

[AA15] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 0.91. Arpaci-Dusseau Books, May 2015.

[AWS] Amazon Web Services, Inc. *Amazon DynamoDB Features*. Last accessed: 29.08.2018. URL: `https://aws.amazon.com/dynamodb/features/`.

[Cho+15] François Chollet et al. *Keras*. Last accessed: 31.08.2018. 2015. URL: `https://keras.io`.

[Chr] Chromium Authors. *src/content/browser/indexed_db/leveldb*. Last accessed: 29.08.2018. URL: `https://cs.chromium.org/chromium/src/content/browser/indexed_db/leveldb/`.

[Cor+09] Thomas H. Cormen et al. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN: 978-0-262-03384-8. URL: `http://mitpress.mit.edu/books/introduction-algorithms-third-edition`.

[CW08] Ronan Collobert and Jason Weston. "A unified architecture for natural language processing: deep neural networks with multitask learning". In: *Machine Learning, Proceedings of the Twenty-Fifth International Conference (ICML 2008), Helsinki, Finland, June 5-9, 2008*. Ed. by William W. Cohen, Andrew McCallum, and Sam T. Roweis. Vol. 307. ACM International Conference Proceeding Series. ACM, 2008, pp. 160–167. ISBN: 978-1-60558-205-4. DOI: `10.1145/1390156.1390177`. URL: `http://doi.acm.org/10.1145/1390156.1390177`.

[DH08] Peter Deuflhard and Andreas Hohmann. *Numerische Mathematik: Eine algorithmisch orientierte Einführung*. 4., überarb. und erw. Aufl. De Gruyter, 2008. ISBN: 978-3-11-020355-4.

[Die+88] Martin Dietzfelbinger et al. "Dynamic Perfect Hashing: Upper and Lower Bounds". In: *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*. IEEE Computer Society, 1988, pp. 524–531. ISBN: 0-8186-0877-3. DOI: `10.1109/SFCS.1988.21968`. URL: `https://doi.org/10.1109/SFCS.1988.21968`.

[Djw17] Djwong. *Ext4 Disk Layout*. Last accessed: 13.04.2018. July 2017. URL: `https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout`.

[Fit+] Brad Fitzpatrick et al. *Memcached – a Distributed Memory Object Caching System*. Last accessed: 29.08.2018. URL: https://memcached.org/.

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. “Storing a Sparse Table with 0(1) Worst Case Access Time”. In: *J. ACM* 31.3 (1984), pp. 538–544. DOI: 10.1145/828.1884. URL: http://doi.acm.org/10.1145/828.1884.

[GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. “Deep Sparse Rectifier Neural Networks”. In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011*. Ed. by Geoffrey J. Gordon, David B. Dunson, and Miroslav Dudík. Vol. 15. JMLR Proceedings. JMLR.org, 2011, pp. 315–323. URL: http://www.jmlr.org/proceedings/papers/v15/glorot11a/glorot11a.pdf.

[GD] Sanjay Ghemawat and Jeff Dean. *LevelDB*. Last accessed: 29.08.2018. URL: https://github.com/google/leveldb.

[Goo05] Google Inc. *sparsehash*. Last accessed: 02.09.2018. 2005. URL: https://github.com/sparsehash/sparsehash.

[He+15] Kaiming He et al. “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”. In: *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*. IEEE Computer Society, 2015, pp. 1026–1034. ISBN: 978-1-4673-8391-2. DOI: 10.1109/ICCV.2015.123. URL: https://doi.org/10.1109/ICCV.2015.123.

[HM95] Jun Han and Claudio Moraga. “The Influence of the Sigmoid Function Parameters on the Speed of Backpropagation Learning”. In: *From Natural to Artificial Neural Computation, International Workshop on Artificial Neural Networks, IWANN ’95, Malaga-Torremolinos, Spain, June 7-9, 1995, Proceedings*. Ed. by José Mira and Francisco Sandoval Hernández. Vol. 930. Lecture Notes in Computer Science. Springer, 1995, pp. 195–201. ISBN: 3-540-59497-3. DOI: 10.1007/3-540-59497-3_175. URL: https://doi.org/10.1007/3-540-59497-3%5C_175.

[Int18] Intel Corporation. *Intel® Math Kernel Library*. 2003–2018. URL: http://software.intel.com/en-us/articles/intel-mkl/.

[Jen97] Robert John Jenkins. “Hash Functions”. In: *Dr. Dobb’s Journal* (1997). URL: http://www.burtleburtle.net/bob/hash/doobs.html.

[JK10] Bob Jenkins and Jozsef Kadlecsik. *jhash.h: Jenkins hash support*. Last accessed: 01.09.2018. 2006, 2009-2010. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/jhash.h?h=v4.18.

[Knu98] Donald Ervin Knuth. *The art of computer programming, Volume III, 2nd Edition.* Addison-Wesley, 1998. ISBN: 0201896850. URL: http://www.worldcat.org/oclc/312994415.

[Kra+18] Tim Kraska et al. "The Case for Learned Index Structures". In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018.* Ed. by Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein. ACM, 2018, pp. 489–504. DOI: 10.1145/3183713.3196909. URL: http://doi.acm.org/10.1145/3183713.3196909.

[KSH17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks". In: *Commun. ACM* 60.6 (2017), pp. 84–90. DOI: 10.1145/3065386. URL: http://doi.acm.org/10.1145/3065386.

[Kuh17] Michael Kuhn. "JULEA: A Flexible Storage Framework for HPC". In: *High Performance Computing - ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P^3MA, VHPC, Visualization at Scale, WOPSSS, Frankfurt, Germany, June 18-22, 2017, Revised Selected Papers.* Ed. by Julian M. Kunkel et al. Vol. 10524. Lecture Notes in Computer Science. Springer, 2017, pp. 712–723. ISBN: 978-3-319-67629-6. DOI: 10.1007/978-3-319-67630-2_51. URL: https://doi.org/10.1007/978-3-319-67630-2%5C_51.

[Lof+16] Jay F. Lofstead et al. "DAOS and friends: a proposal for an exascale storage system". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016.* Ed. by John West and Cherri M. Pancake. IEEE Computer Society, 2016, pp. 585–596. ISBN: 978-1-4673-8815-3. DOI: 10.1109/SC.2016.49. URL: https://doi.org/10.1109/SC.2016.49.

[LS07] Pierre L'Ecuyer and Richard J. Simard. "TestU01: A C library for empirical testing of random number generators". In: *ACM Trans. Math. Softw.* 33.4 (2007), 22:1–22:40. DOI: 10.1145/1268776.1268777. URL: http://doi.acm.org/10.1145/1268776.1268777.

[Lüt+18] Jakob Lüttgau et al. "Survey of Storage Systems for High-Performance Computing". In: *Supercomputing Frontiers and Innovations* 5.1 (2018). ISSN: 2313-8734. URL: http://superfri.org/superfri/article/view/162.

[Mar+15] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* Software available from tensorflow.org, Last accesed: 31.08.2018. 2015. URL: https://www.tensorflow.org/.

[MP43] Warren S. McCulloch and Walter Pitts. “A Logical Calculus of the Ideas Immanent in Nervous Activity”. In: *Bulletin of Mathematical Biophysics* 5 (1943), pp. 115–133.

[Oli06] Travis E. Oliphant. *A Guide to NumPy*. USA: Trelgol Publishing, 2006.

[Rus+15] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y. URL: https://doi.org/10.1007/s11263-015-0816-y.

[Sto09a] Jorge Stolfi. *File:Hash table 5 0 1 1 1 1 0 LL.svg — Wikimedia Commons, the free media repository*. Last accessed: 24.08.2018. 2009. URL: https://commons.wikimedia.org/w/index.php?title=File:Hash_table_5_0_1_1_1_1_0_LL.svg&oldid=205101359.

[Sto09b] Jorge Stolfi. *File:Hash table 5 0 1 1 1 1 0 SP.svg — Wikimedia Commons, the free media repository*. Last accessed: 24.08.2018. 2009. URL: https://commons.wikimedia.org/w/index.php?title=File:Hash_table_5_0_1_1_1_1_0_SP.svg&oldid=205101350.

[Sto09c] Jorge Stolfi. *File:Hash table 5 0 1 1 1 1 1 LL.svg — Wikimedia Commons, the free media repository*. Last accessed: 24.08.2018. 2009. URL: https://commons.wikimedia.org/w/index.php?title=File:Hash_table_5_0_1_1_1_1_1_LL.svg&oldid=205101361.

[Sym] Symas Corporation. *Lightning Memory-mapped Database*. Last accessed: 29.08.2018. URL: https://symas.com/lmdb/.

[Tan15] Zilong Tan. *fast-hash*. Last accessed: 31.08.2018. 2015. URL: https://github.com/ZilongTan/Coding/tree/master/fast-hash.

[The18] The HDF Group. *Hierarchical Data Format, version 5*. Last accessed: 31.08.2018. 1997-2018. URL: http://www.hdfgroup.org/HDF5/.

[Uni18] Unidata. *Network Common Data Form (NetCDF), version 4*. Last accessed: 31.08.2018. 2008-2018. URL: https://doi.org/10.5065/D6H70CW6.

[Xia+18] Zhang Xianyi et al. *OpenBLAS: An optimized BLAS library*. Last accessed: 09.09.2018. 2011–2018. URL: https://www.openblas.net/.

Appendices

A. Additional Measurements

This chapter contains additional measurements to the ones in Chapter 5.

A.1. Distribution of Entries to Subnets

A.1.1. Dataset A

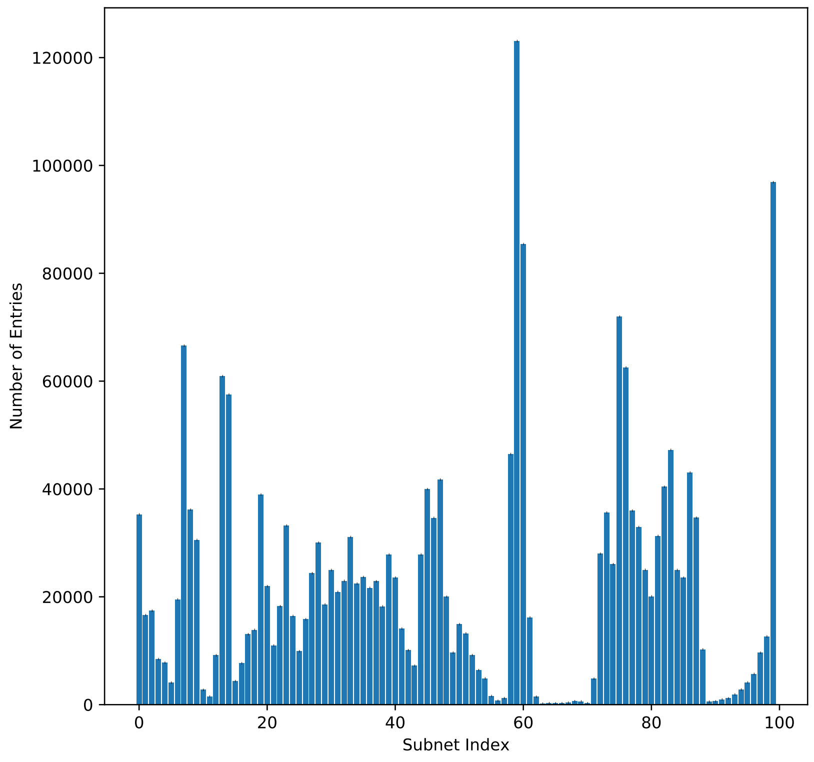


Figure A.1.: Number of entries per subnet for 100 subnets and a first stage with 16 nodes in the hidden layer, trained for 64 epochs. First-stage loss: `0.00125`, total loss: `0.000159`.

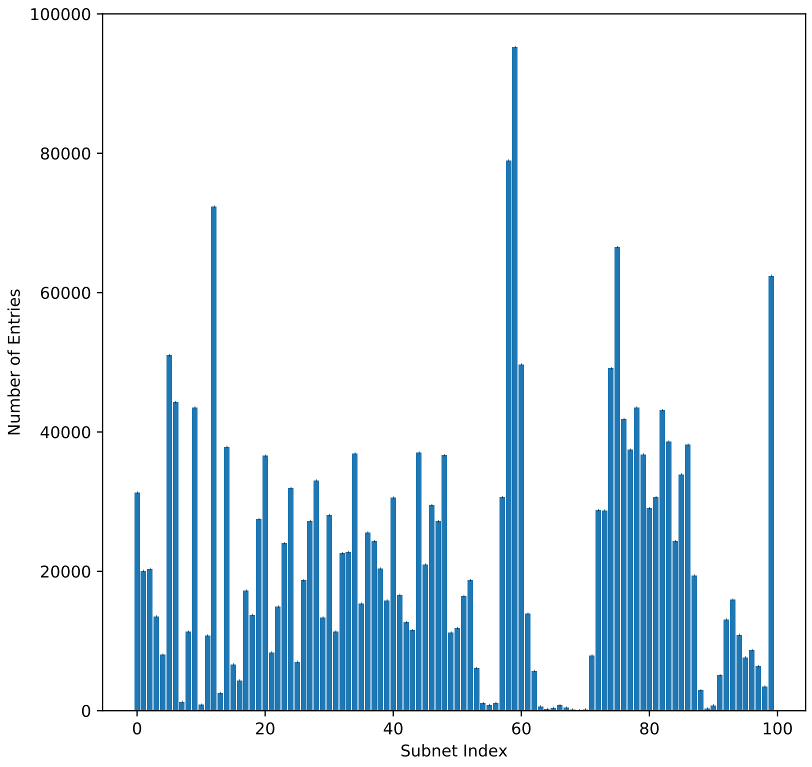


Figure A.2.: Number of entries per subnet for 100 subnets and a first stage with 16 nodes in the hidden layer, trained for 128 epochs. First-stage loss: `0.00111`, total loss: `0.000116`.

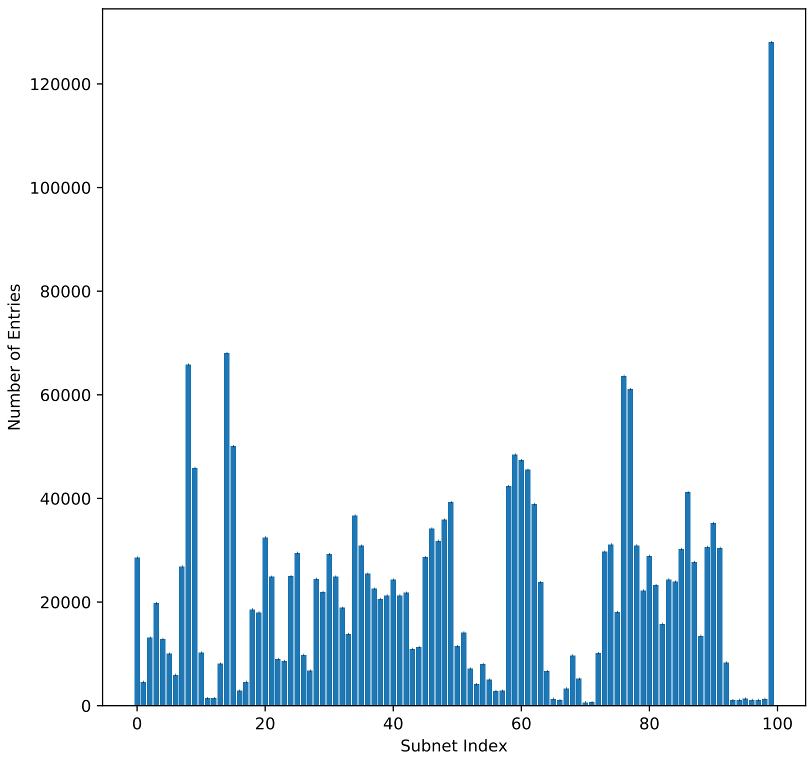


Figure A.3.: Number of entries per subnet for 100 subnets and a different first stage also with 16 nodes in the hidden layer, trained for 128 epochs. First-stage loss: `0.00112`, total loss: `0.000112`.

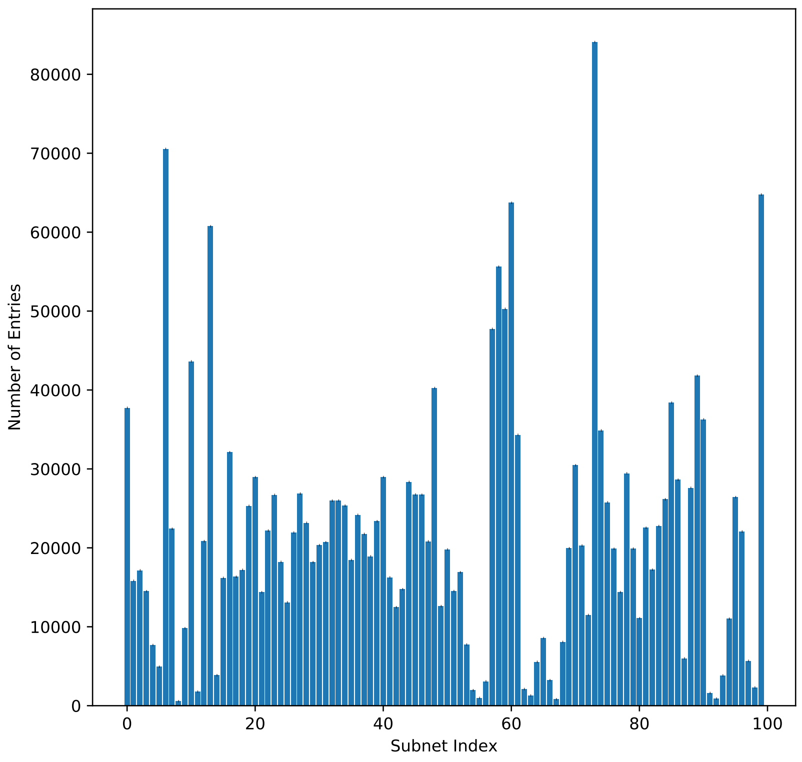


Figure A.4.: Number of entries per subnet for 100 subnets and a first stage with 256 nodes in the hidden layer, trained for 128 epochs. First-stage loss: `0.000567`, total loss: `5.756e-5`.

A.1.2. Dataset I

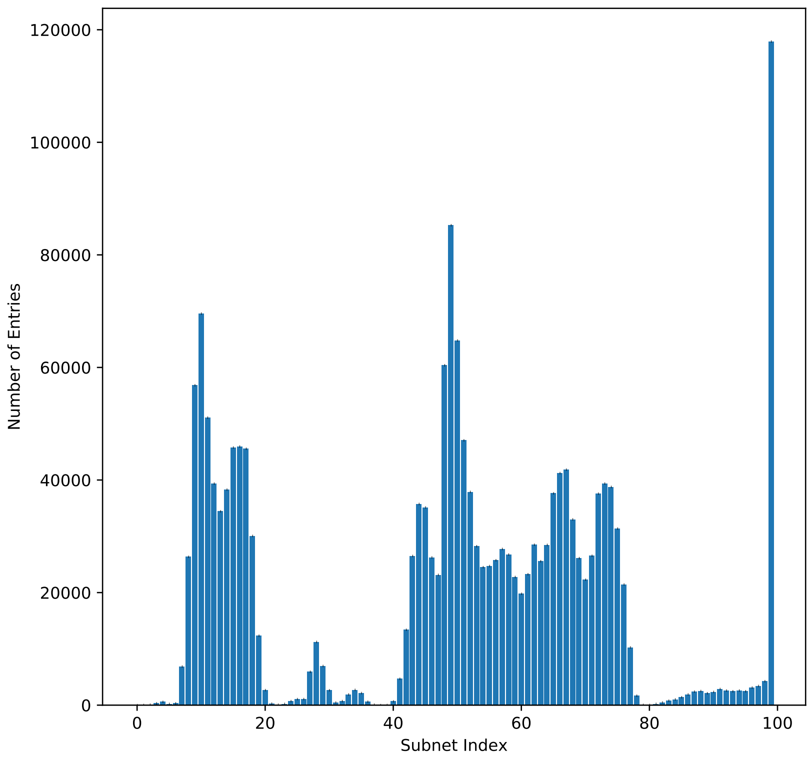


Figure A.5.: Number of entries per subnet for 100 subnets and a first stage with zero hidden layers. First-stage loss: `0.00712`, total loss: `6.202e-5`.

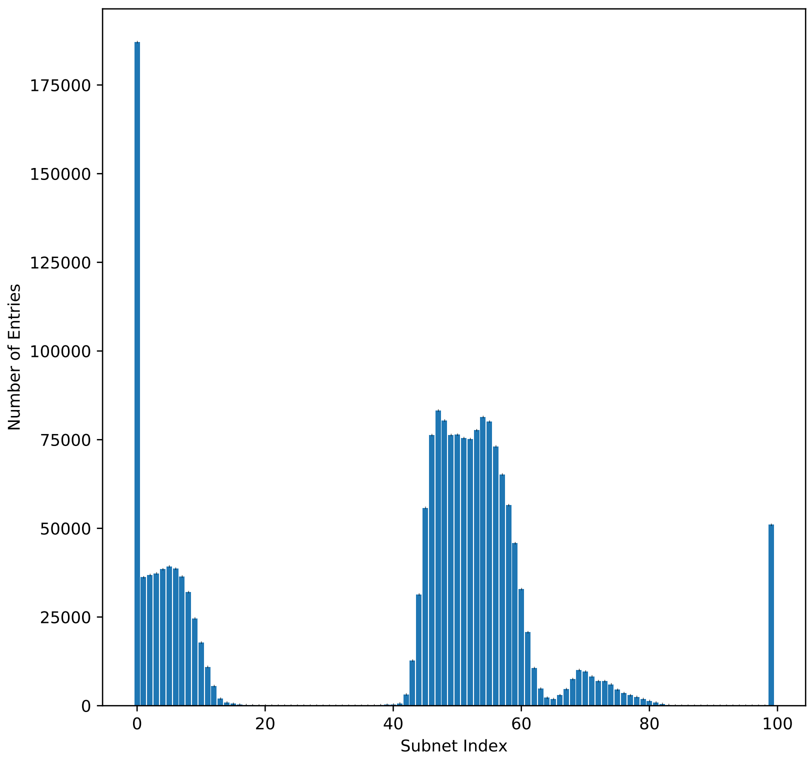


Figure A.6.: Number of entries per subnet for 100 subnets and a first stage with 16 nodes in the hidden layer, trained for 4 epochs. First-stage loss: `0.0269`, total loss: `0.000612`.

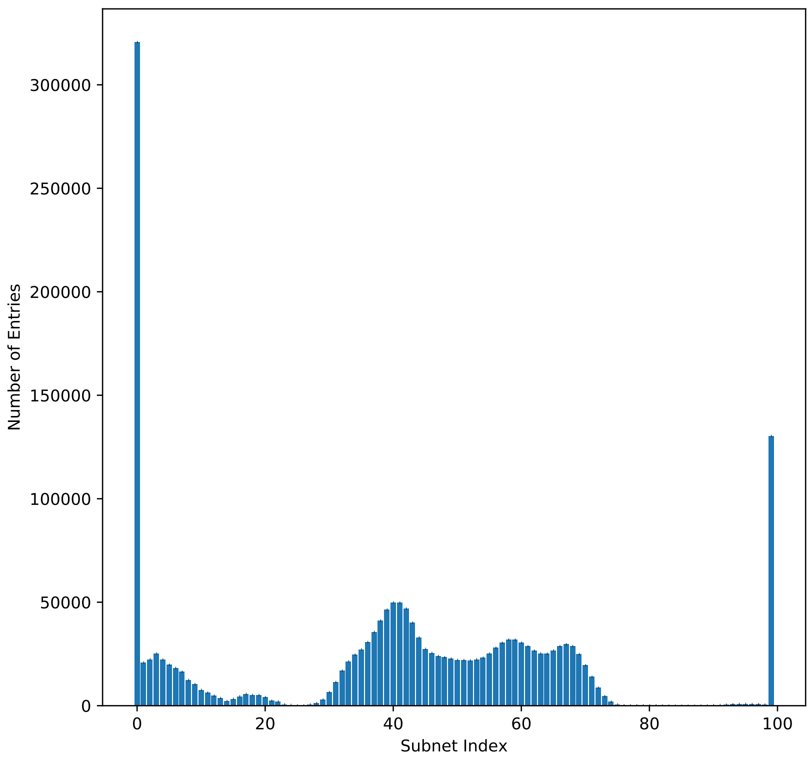


Figure A.7.: Number of entries per subnet for 100 subnets and a first stage with 16 nodes in the hidden layer, trained for 16 epochs. First-stage loss: `0.0174`, total loss: `0.000166`.

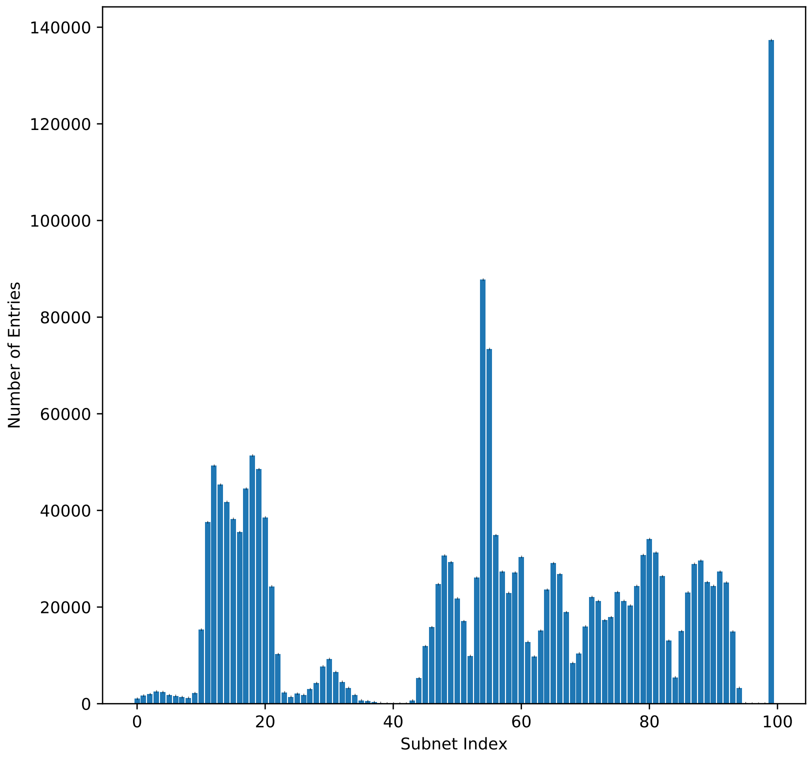


Figure A.8.: Number of entries per subnet for 100 subnets and a first stage with 32 nodes in the hidden layer, trained for 16 epochs. First-stage loss: `0.00822`, total loss: `5.972e-5`.

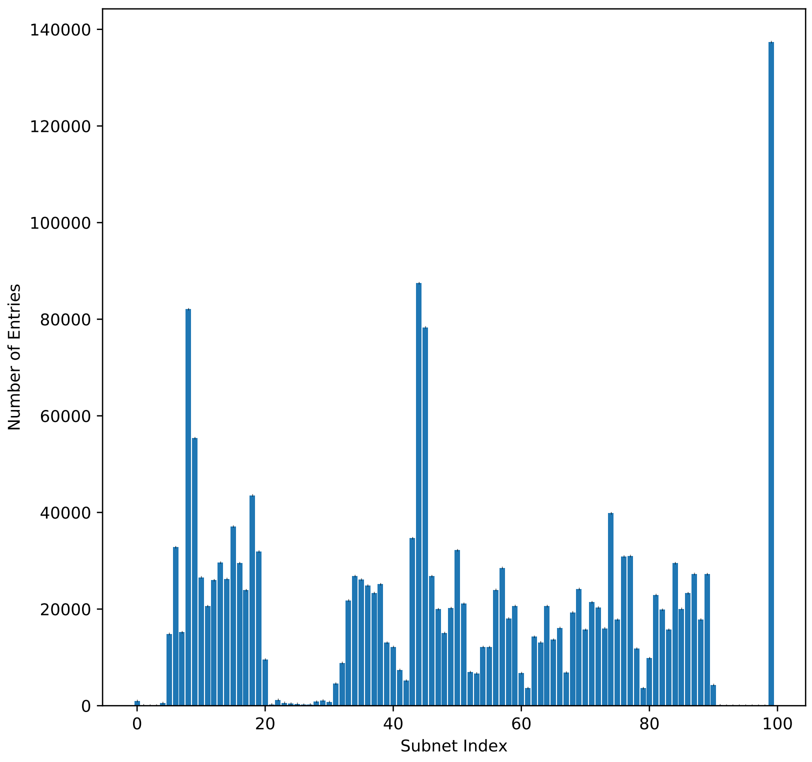


Figure A.9.: Number of entries per subnet for 100 subnets and a first stage with 64 nodes in the hidden layer, trained for 64 epochs. First-stage loss: `0.00314`, total loss: `2.405e-5`.

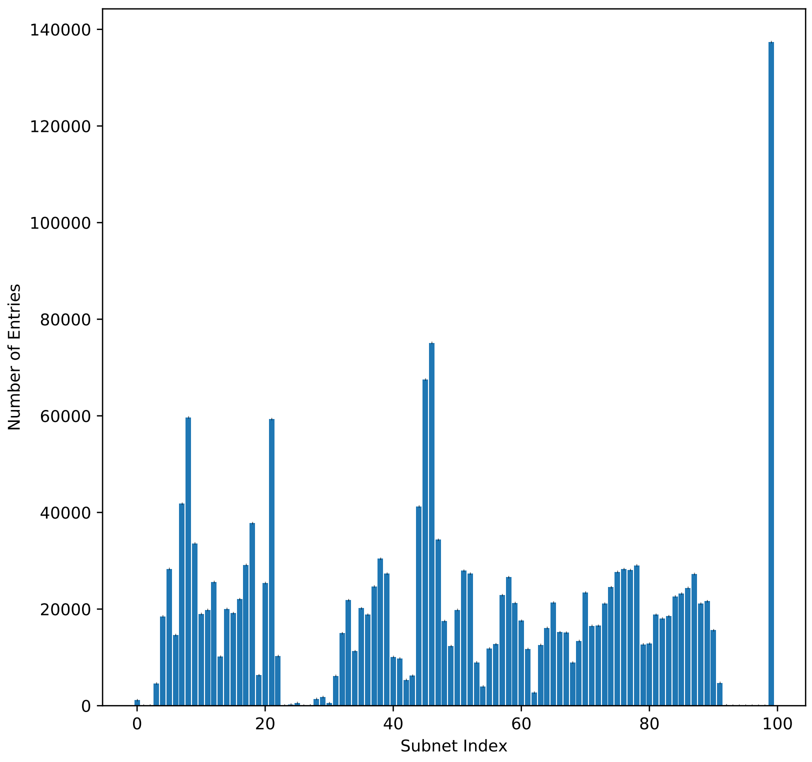


Figure A.10.: Number of entries per subnet for 100 subnets and a first stage with 128 nodes in the hidden layer, trained for 128 epochs. First-stage loss: `0.00226`, total loss: `2.187e-5`.

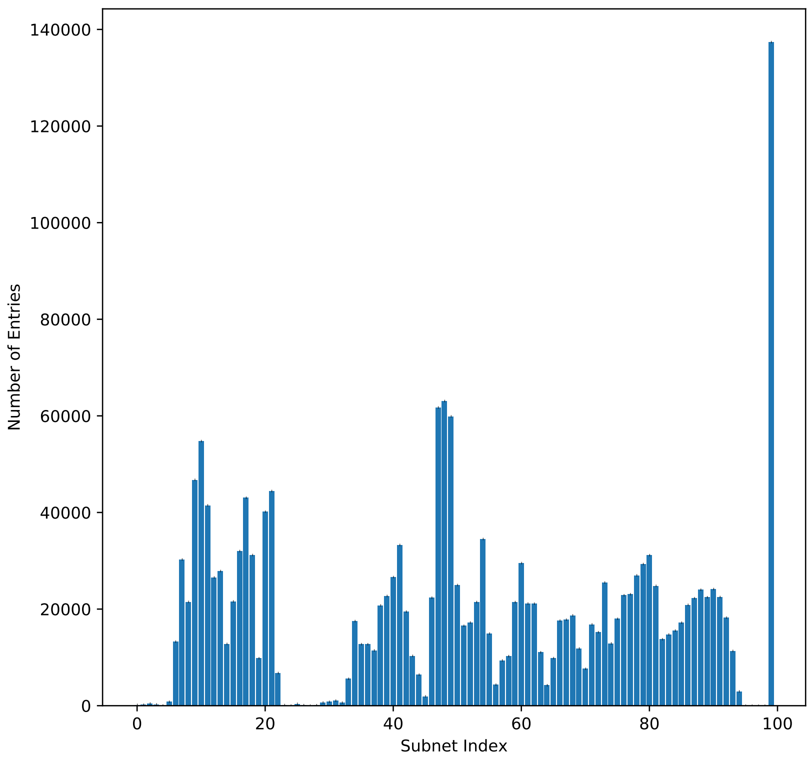


Figure A.11.: Number of entries per subnet for 100 subnets and a first stage with 256 nodes in the hidden layer, trained for 128 epochs. First-stage loss: `0.00300`, total loss: `2.394e-5`.

A.2. Number of Subnets

Number of Subnets	One Hidden Layer	Zero Hidden Layers
0	0.00395	0.00329
10	0.00160	0.00159
50	0.00111	0.000883
100	0.000991	0.000810
500	0.000564	0.000551
1000	0.000353	0.000339
5000	0.000322	0.000305
10000	0.000314	0.000301
50000	0.000284	0.000285
100000	0.000253	0.000267

Table A.1.: Overall loss in relation to the number of subnets. Zero subnets means that the loss of the first stage was taken.

List of Figures

List of Listings

List of Tables

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Bachelorstudiengang Computing in Science selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ort, Datum	Unterschrift

Veröffentlichung

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Ort, Datum	Unterschrift